METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION
(del 16 de abril al 15 de mayo de 1982)

DIA	HORARIO	TEMAS	PROFESORES
16 de abril	17:00 a 18:00	EL ENFOQUE ESTRUCTURADO DE DESARROLLO DE SOFTWARE	Ing. Jorge Euan Avila
	18:00 a 21:00	ANALISIS ESTRUCTURADO	
17 de abril	9:00 a 14:00	ANALISIS ESTRUCTURADO	Ing. Jorge Euan Avila
23 de abril	17:00 a 21:00	DISEÑO ESTRUCTURADO	Ing. Alejandro Acosta
24 de abril	9:00 a 14:00	DISEÑO ESTRUCTURADO	Ing. Alejandro Acosta
30 de abril	17:00 a 21:00	PROGRAMACION ESTRUCTURADA	Ing. Raymundo H. Rangel Gutiérrez
7 de mayo	17:00 a 21:00	PROGRAMACION ESTRUCTURADA	Ing. Raymundo H. Rangel Gutiérrez
8 de mayo	9:00 a 14:00	CONTROL DE CALIDAD	M. en C. Marcial Portilla Robertson
14 de mayo	17:00 a 21:00	MANTENIMIENTO DE PROGR.	Ing. Armando Díaz Espejal
15 de mayo	9:00 a 14:00	CASOS PRACTICOS	Ing. Alejandro Acosta Ing. José Origel Coutiño

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS

DE INFORMACION

MANTENIMIENTO DE PROGRAMAS

ING. ARMANDO DIAZ

MAYO, 1982

THE CONTEXT OF DESIGN

Peter Freeman
University of California, Irvine

Before undertaking a study of software design techniques, it is important to understand the total context in which software design takes place. This paper examines the role of design in the software development life cycle and illustrates the distinctions between design and programming.

THE LIFE CYCLE OF SOFTWARE DEVELOPMENT

Although there are many characterizations of the system development process (see, for example, Teichroew [2] or many of the entries in Section VI), we believe the following five stage cycle provides a good basis for our discussions:

1) Needs Analysis - The eventual user of a system discovers a need for which an information system seems to be the answer; the nature of the need is analyzed; the outlines of the type of system that would satisfy these needs are established. This may be a formal stage in the process or it may occur concurrently with the specification stage.

2) Specification - Functional descriptions of the system are developed, along with constraints on its structure and resource usage. Economic constraints on the development process itself are stated. This stage may be combined with the needs analysis and in many instances is blended into the design stages in an interative sequence that goes back and forth between statement of specifications and refinement of the design.

3) Architectural Design - Working from the specifications, the underlying structure of the problem for which this system will be a solution is sought out. When this structure becomes clear, a design of the system is devised. This design is necessarily at a gross level of detail. The parts of the system and their relationships, the basic algorithms that the system will use, and the major data representations that will be needed are all primary elements of the design at this stage.

4) Detailed Design - The parts of the design are now more detailed. Precise algorithms and data structures are spelled out. Interfaces between parts are detailed. If not already done at the architectrual stage, decisions as to which parts will be software and which will be hardware are made. Detailed design (as well as architectural design), may require several levels of refinement. This stage stops short of spelling out all programming details (e.g. housekeeping, local data structures).

5) Implementation - Producing a physical realization of the design includes programming, testing of individual pieces, integration of pieces into subassemblies, system testing, performance evaluation. In many cases, a good deal of redesign and reimplementation may take place at this stage to make the actual system conform to the specifications and initial requirements. (If we were investigating all parts of the cycle, not just design, then this stage should be broken into three stages: programming, testing, and system integration).

6) Maintenance - Everything that happens after the system is "finished" is termed maintenance: location and repair of bugs, addition of new functions, modification of existing functions, addition of features originally in the design but never implemented.

It is important to recognize two things that so far have been left implicit. First, the stages we have outlined here are temporal stages. Although they are named by the dominant activity of each stage, as we will see below, many activities may occur at each stage. Thus, for example, design may occur during a maintenance activity consisting of the addition of a new function.

Second, we have not clearly differentiated between software design and total system (i.e. hardware plus software) design. This was intentional. In some situations, there is little or no choice of hardware, nor of which functions will be in hardware and which will be in software. In such cases, most of the stages outlined above deal primarily with software design. In other cases, the choice of hardware may be open and the focus of the design effort down to a relatively low level may be on the logical design of the total system, with decisions between hardware and software implementation delayed as long as possible.

The overall structure of the development process in either case is nearly the same, even though the nature of the decisions made varies. Thus, for our purposes here we have not made a careful distinction.

ACTIVITIES AT EACH STAGE

In order to understand better what must be done to create a system, an informal system analysis of the development process will be helpful. For each stage, we will list the primary inputs, outputs, and major operations. There are, of course, some inputs (such as the general knowledge of the people involved) which are present at all stages, but we will not list these explicitly.

Needs Analysis

I: primitive needs, system context, user problems

O: general requirements

OP: identification of major functions and constraints

Specification

I: requirements, system analysis of context

O: specifications of system functions, constraints, and objectives

OP: conversion of needs into explicit functions, selection of constraints that are operational

Architectural Design

I: specifications, general context of desired system, knowledge of similar systems

O: structural description of system

OP: discovery of problem structure, identification of major pieces of system, establishment of relationships between parts, abstraction

Detailed Design

I: architectural description, programming environment details

O: blueprints for programs

OP: abstraction, elaboration, choice of alternatives

Implementation

I: blueprints

O: program code, data and file layouts, working system

OP: encoding of algorithms and data representations, testing, debugging

Maintenance

I: system, documentation, operational requirements

O: improved system

OP: debugging, redesign, reprogramming, enhancement

The most important point to be from this informal analysis is that ... takes place during many different ... the development cycle. While it is ... to identify some stages as ... design-intensive, the application of desi techniques may take place at many points.

So much software development has been done in the past without the benefit of much logical design, development of small programs often does not require much design) many people tend to confuse programming with design. (Indeed, the development of small programs often does not require much design). While there is a close interaction between the two activities, they are different. To emphasize this, we will now look briefly at their differences.

DESIGN VS. PROGRAMMING

If we characterize the activities in software design and development in a general way, it is clear that programming is only part of the total process. First, consider the several activities involved in programming at the lowest level (coding):

-devising local and concrete data representations for information;

-forming precise algorithms for doing necessary processing;

-taking care of housekeeping details necessitated by the particular programming system used (language plus run-time environment);

-choosing names, forming syntactically correct language statements, and making the program letter perfect.

By contrast, design, especially at the higher levels, is concerned with somewhat different activities:

-abstracting the operations and data of the task situation so that they may be represented in the system;

-determining precisely what is to be done by the software under design;

-establishing an overall structure of the system;

-establishing interfaces and definite control and data linkages between parts of the system and between the system and other systems;

-choosing between major design alternatives;

-making tradeoffs dictated by global constraints and conditions in order to meet varied requirements such as reliability,

generality, user-centeredness, and so on.

3

There is an easy way of distinguishing between the two activities. When programming, we are constructing programs (often in some higher-level language). We must bind data representations and control sequences so that the program will execute properly. When designing, we are devising representations of programs, not actual programs. Clearly, we will be doing some of the same things in both activities -- we may choose some definite control sequences during design, for example -- and detailed design often comes arbitrarily close to producing complete programs (especially when using a metacode).

Another way of emphasizing the difference between programming and design is to consider what is actually produced -- that is, what is on the pieces of paper produced by a programmer or a designer. A programmer produces programs and documentation of those programs. The documentation is needed to make the programs understandable to humans but it is not a necessary condition of existence. That is, the programs exist without the documentation, they may execute without the documentation, and indeed we often instruct someone studying a program to "go to the code" for the final word on program content.

By contrast, when designing we produce representations of programs. These representations may be very high-level in detail (e.g. an overall black-box diagram representing major control-flow) or very near to actual programs (e.g. a metacode description of a program that is complete except for minor housekeeping details).

Thus, design is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher levels and on the logical operations involved at the lower levels. Programming is then the process of turning such a representation into a program that will execute on a specific machine.

WHY DESIGN?

Over the past few years, the need for systematic software design has received increasingly widespread acceptance. Therefore, it is is unnecessary to go to great lengths to motivate the study of design techniques. However, a brief discussion of a few major factors will help to place design in context.

Perhaps the most important reason to design is that the creation of complex systems involves a very large amount of detail and complexity (i.e. relationships of many sorts between many of the pieces). If this complexity is not controlled, then the desired results will rarely be achieved. Design is the primary tool for controlling and dealing with this mass of detail and attendant complexity. The abstracting processes of design permit us to deal with large masses of detail without becoming bogged down. The regularity and structure of design methods and techniques serve to guide us through complex chains of reasoning where we might otherwise become lost.

A second important reason to design is to aid in the discovery of the underlying structure of the problem situation. An artifact created for any purpose must fit itself to the environment in which it will exist if it is to be successful. In simple situations, the nature of the environment and the necessary structure of the artifact may be more or less obvious and/or prescribed. In complex situations this is rarely the case and this structure must be discovered through design.

A third and increasingly important reason for design is its impact on system quality. We want systems that are reliable, user-centered, efficient, portable, and so on. These are all properties of a system that are global in nature and which demand global design decisions.

During design we are working with logical properties and the overall structure of the system. There is not yet a huge investment in code and detailed decisions which cannot be changed when an evaluation indicates that desired system properties are not being met. If reliability, useful user functions, modularity, and so on are not planned for before programming is begun, then generally they will be unobtainable. Design at a logical level provides the flexibility to make the global decisions needed to achieve desired system quality.

CONCLUSION

It is impossible in a short paper to describe all possible software design situations. The design problems you face, the organizational context in which you work, and your personal skills will vary in some respects from what we have described here. However, this introduction, coupled with the other material and references in this volume, should provide you enough global view to begin the improvement of your design skills.

REFERENCES

1. Freeman, Peter. "Software Design Representation: Analysis and Improvements," TR81, ICS Department, University of California, Irvine, 1976.

2. Teichroew, Dan. "Improvements in the System Life Cycle." Proc. 1974 IFIP Congress, North-Holland Publishing Co., pp. 972-978.

One would expect that current information-processing research and development projects would be strongly oriented toward where the future problems are. However, according to recent Congressional testimony by Dr. Ruth Davis of the National Bureau of Standards (NBS) on federally-funded computing R&D projects:

> . . . 21% of the projects were concerned with hardware design, 40% were concerned with the needs of special interest communities such as natural sciences, engineering, social and behavioral sciences, humanities, and real-time systems. 14% were in the long-range payoff areas of metatheory, while only 9% were oriented to the highly agonizing software problems identified by most customers as their major concern."[4]

One result of the CCIP-85 study has been to begin to reorient Air Force information processing R&D much more toward software. Similar R&D trends are evident at DOD's Advanced Research Projects Agency (ARPA), National Science Foundation, and the National Bureau of Standards. But much remains to be done.

Indirect costs even bigger

Big as the direct costs of software are, the indirect costs are even bigger, because software generally is on the critical path in overall system development. That is, any slippages in the software schedule translate directly into slippages in the overall delivery schedule of the system.

Let's see what this meant in a recent software development for a large defense system. It was planned to have an operational lifetime of seven years and a total cost of about $1.4 billion—or about $200 million a year worth of capability. However, a six-month software delay caused a six-month delay in making the system available to the user, who thus lost about $100 million worth of needed capability—about 50 times the direct cost of $2 million for the additional software effort. Moreover, in order to keep the software from causing further delays, several important functions were not provided in the initial delivery to the user.

Again, similar situations develop in domestic applications. IBM's OS 360 software was over a year late.[5] The U.S. air traffic control system currently operates much more expensively and less effectively because of slippages of years in software (and also hardware, in this case) development, which have escalated direct software costs to over $100 million.[6] Often, organizations compensate for software development slippages by switching to a new system before the software is adequately tested, leading to such social costs as undelivered welfare checks to families with dependent children, bad credit reports, and even people losing their lives because of errors in medical software.

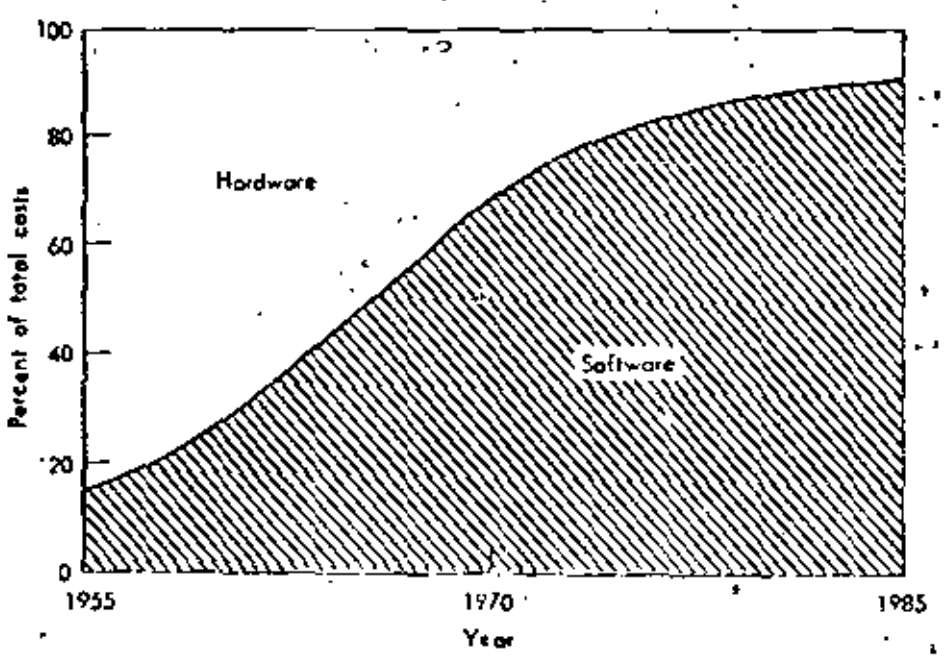


Fig. 6. Hardware/software cost trends.

Getting software off the critical path

Once software starts slipping along the critical path, there are several more or less unattractive options. One option is to add more people in hopes that a human wave of programmers will quickly subdue the problem. However, Brooks' excellent article[7] effectively shows that software is virtually incompressible with respect to elapsed time, and that such measures more often make things worse rather than better. Some other unhappy options are to skimp on testing, integration, or documentation. These usually cost much more in the long run. Another is just to scrap the new system and make do with the old one. Generally, the most attractive option is to reduce the system to an austere but expandable initial capability.

For the future, however, several opportunities exist for reducing software delays and getting software off the critical path. These fall into three main categories:

1. Increasing each individual's software productivity.

2. Improving project organization and management.

3. Initiating software development earlier in the system development cycle.

[4] "Government Bureau Takes on Role of Public Protector Against Computer Misuse," *Communications of the ACM*, November 1972, p. 1018.

[5] Hirsch, R., "What's Wrong With the Air Traffic Control System?" *Datamation*, August 1972, pp. 48-51.

[6] Remkis, F., "Why Is The Software Late?" *Data Management*, August 1971.

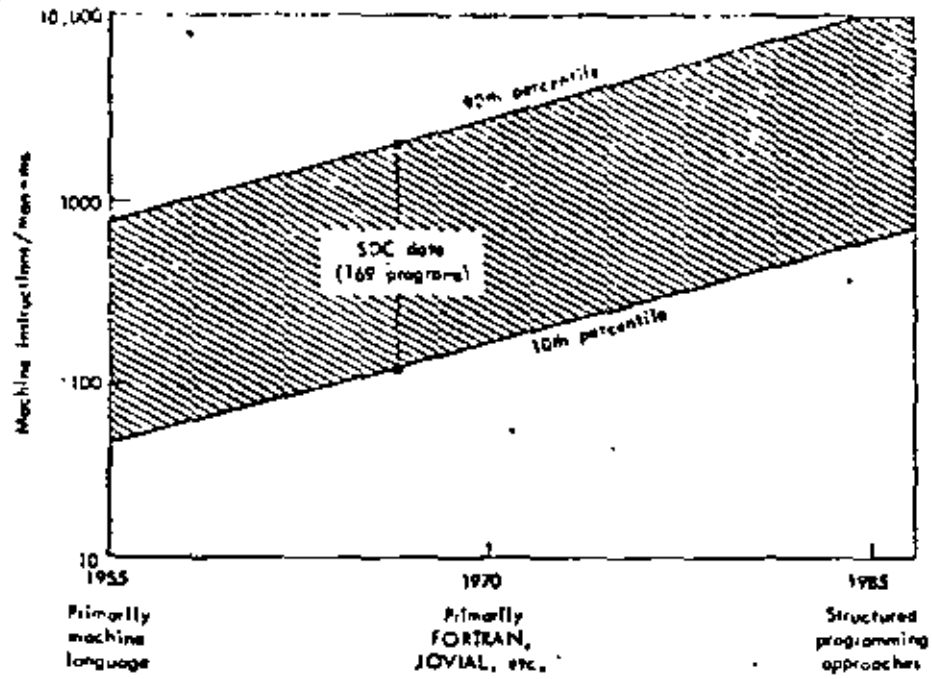


Fig. 7. Technology forecast: software productivity.

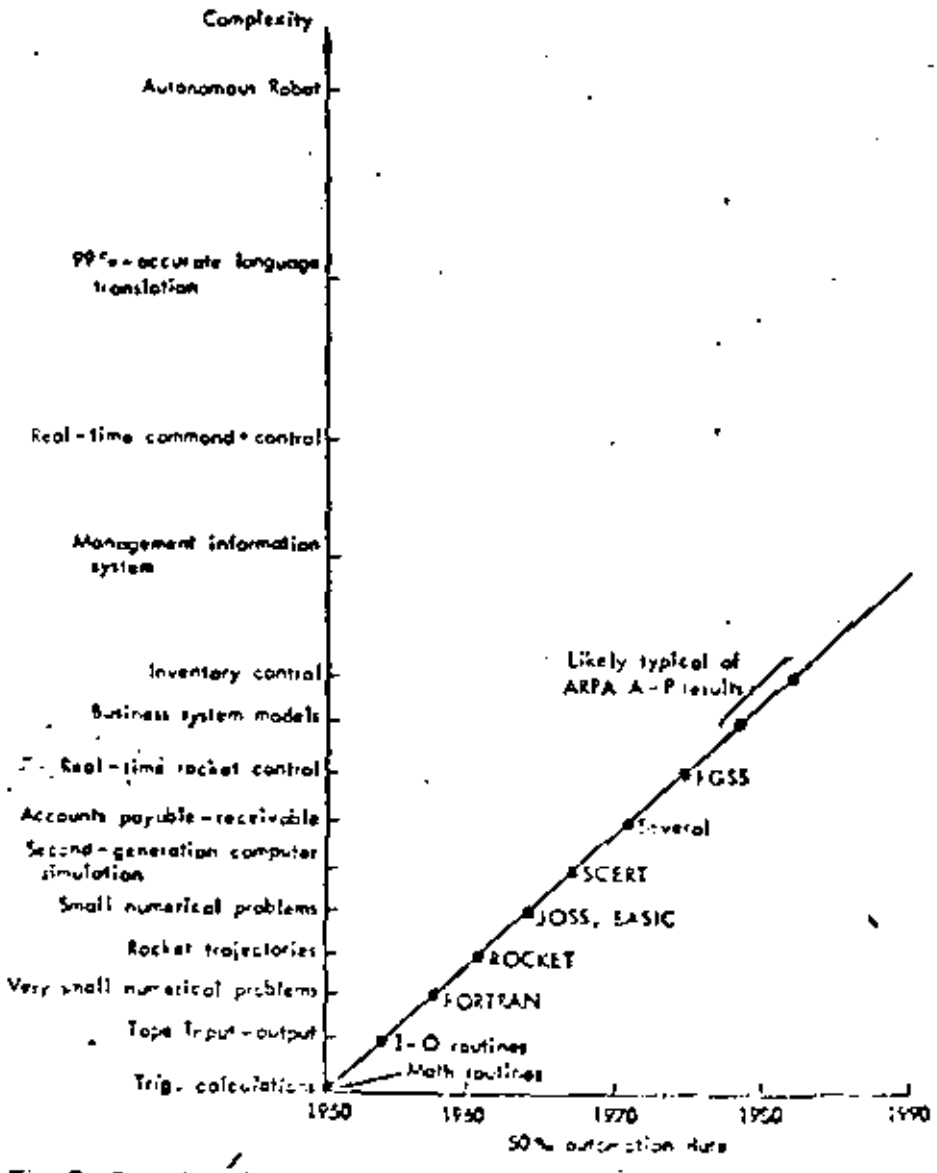


Fig. 8. Growth of automatic programming.

Software Impact

Increasing software productivity: definitions

Fig. 7 shows a simplistic view of likely future trends in software productivity. It is probably realistic in maintaining at least a factor-of-10 spread between the 10th and 90th percentiles of software productivity, but it begs a few important questions.

One is, "What is software?" Even the courts and the Internal Revenue Service have not been able to define its metes and bounds precisely. The figures above include computer program documentation, but exclude operating procedures and broad system analysis. Clearly, a different definition would affect software productivity figures significantly.

Another important question is, "What constitutes software production?" As early as the mid-1950s there were general-purpose trajectory analysis systems with which an analyst could put together a modular, 10,000-word applications program in about 10 minutes. Was this "software production?" With time, more and more such general-purpose packages as ICES (MIT's Integrated Civil Engineering System), Programming-by-Questionnaire, RPG, MARK IV, and SCERT have made the creation of significant software capabilities so easy that they tend to be eliminated from the category of "software productivity," which continues to refer to those portions of the software directly resulting from handwritten strings of assembly or FORTRAN-level language statements. Fig. 8 is an attempt to characterize this trend in terms of a "50% automation date": the year in which most of the incoming problems in an area could be "programmed" in less than an hour by a user knowledgeable in his field, with one day of specialized training.

Thus, if we want to speak objectively about software productivity, we are faced with the dilemma of:

1. Either redefining it in terms of source instructions rather than object instructions—thereby further debasing the unit of production (which isn't completely objective even using object instructions as a base)—or

2. Continuing to narrow the range of definition of "software productivity" to the more and more difficult programs which can't be put together more or less automatically.

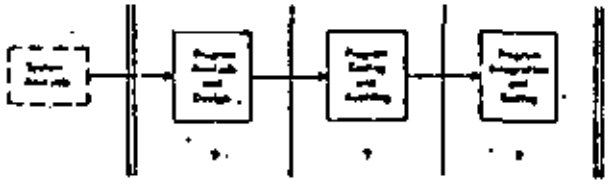
The eventual result of ARPA's major "automatic programming" effort will be to narrow this latter range even further.[1]

[1] Balzer, Robert M., [illegible], Information Sciences Institute Technical Memorandum, University of Southern California, Information Sciences Institute, September 1972.

A Software Quiz

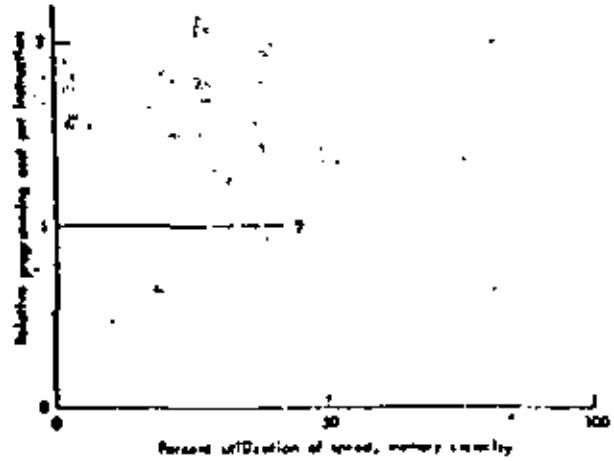
Very little in the way of quantitative data has been collected about software. But there is some which deserves to be better known than it is. Because, otherwise, we have nothing but our intuition to guide us in making critical decisions about software, and often our intuition can be quite fallible. The four questions below give you a chance to test how infallible your software intuition is. Answers to the quiz appear on the following two pages.

1A. Where Does the Software Effort Go?



If you're involved in planning, staffing, scheduling or integrating a large software effort, you should have a good idea of how much of the effort will be spent on analysis and design (after the functional specification for the system has been completed), on coding and auditing (including desk checking and software module unit testing), and on checkout and test. See how well you do in estimating the effort on a percentage basis for the three phases. The results for such different large systems as SAGE, OS/360, and the Gemini space shots have been strikingly similar.

2A. How Do Hardware Constraints Affect Software Productivity?



Percent utilization of speed, memory capacity

Another useful factor to know in planning software development is the extent to which hardware constraints affect software productivity. As you approach complete utilization of hardware speed and memory capacity, what happens to your software costs? Do they stay relatively constant or do they begin to bulge upward somewhat? The data here represent 34 software projects at North American Rockwell's Autonetics Division with some corroborative data points determined at Mitre.

3A. Where Are Software Errors Made?

	Batch (all errors)	Real-time (final validation phase only)
Computation and assignment	?	?
Sequencing and control	?	?
Input-output	?	?
Declarations	?	?
Punctuation	?	Not available
Correction to errors	Not available	?
Total	100%	100%

If you're setting test plan schedules and priorities, designing diagnostic aids for compilers and operating systems, or contemplating new language features (e.g., GOTO-free) to eliminate sources of software errors, it would be very useful to know how such errors are distributed over the various software functions. See how well you do in estimating the distribution of errors for typical batch programs and for the final validation of a critical real-time program.

4A. How Do Compilers Spend Their Time?

(Knuth study: 440 Lockheed programs; 250,000 statements)

Number of operands	%
1 (A = B)	?
2 (A = B ⊙ C)	?
3 (A = B ⊙ C ⊙ D)	?
>3	?

Recently, Donald Knuth and others at Stanford performed a study on the distribution of complexity of FORTRAN statements. Try to estimate what percentage of their sample of 250,000 FORTRAN statements were of the simple form A=B, how many had two operands on the right-hand side, etc. If you're a compiler designer, this should be very important, because it would tell you how to optimize your compiler—whether it should do simple things well or whether it should do complex things well. Here the results refer to aerospace application programs at Lockheed; however, a sample of Stanford student programs showed roughly similar results.

Software and Its Impact: A Quantitative Assessment

by Barry W. Boehm

"You software guys are too much like the weavers in the story about the Emperor and his new clothes. When I go out to check on a software development the answers I get sound like, 'We're fantastically busy weaving this magic cloth. Just wait a while and it'll look terrific.' But there's nothing I can see or touch, no numbers I can relate to, no way to pick up signals that things aren't really all that great. And there are too many people I know who have come out at the end wearing a bunch of expensive rags or nothing at all."

—An Air Force decisionmaker

•
•
•
•
•

Recently, the Air Force Systems Command* completed a study, "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s," or CCIP-85 for short. The study projected future Air Force command and control information processing requirements and likely future information processing capabilities into the 1980s, and developed an Air Force R&D plan to correct the mismatches found between likely capabilities and needs.

Although many of the CCIP-85 conclusions are specific to the Air Force, there are a number of points which hold at least as well elsewhere. This article summarizes those transferable facts and conclusions.

Basically, the study showed that for almost all applications, software (as opposed to computer hardware, displays, architecture, etc.) was "the tall pole in the tent"—the major source of difficult future problems and operational performance penalties. However, we found it difficult to convince people outside the software business of this. This was primarily because of the scarcity of solid quantitative data to demonstrate the impact of software on operational performance or to provide perspective on R&D priorities.

The study did find and develop some data which helped illuminate the problems and convince people that the problems were significant. Surprisingly, though, we found that these data are almost unknown even to software practitioners. (You can test this assertion via the Software Quiz, p. 51.) The main purpose of this article is to make these scanty but important data and their implications better known, and to convince people to collect more of it.

Before reading further, though, please try the Software Quiz. It's intended to help you better appreciate the software issues which the article goes on to discuss.

Software is big business

One convincing impact of software is directly on the pocketbook. For the Air Force, the estimated dollars for FY 1972 are in Fig. 5: an annual expenditure on software of between $1 billion and $1.5 billion, about three times the annual expenditure on computer hardware and about 4 to 5% of the total Air Force budget. Similar figures hold elsewhere. The recent World Wide Military Command and Control System (WWMCCS) computer procurement was estimated to involve expenditures of $50 to $100 million for hardware and $722 million for software.[1] A recent estimate for NASA was an annual expenditure of $100 million for hardware, and $200 million for software — about 6% of the annual NASA budget.

For some individual projects, here are some overall software costs:

IBM OS/360	$ 200,000,000[2]
SAGE	250,000,000[3]
Manned Space Program, 1960-70	1,000,000,000[4]

Overall software costs in the U.S. are probably over $10 billion per year, over 1% of the gross national product.

If the software-hardware cost ratio appears lopsided now, consider what will happen in the years ahead, as hardware gets cheaper and software (people) costs go up and up. Fig. 6 shows the estimate for software expenditures in the Air Force going to over 90% of total adp system costs by 1985: this trend is probably characteristic of other organizations, also.

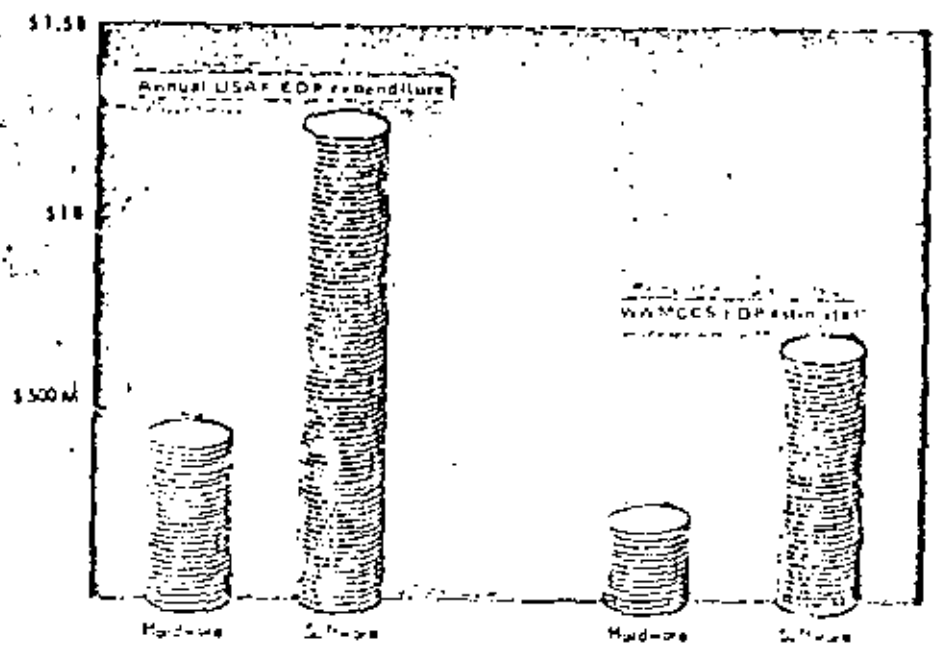


Fig. 5. USAF software is big business.

*The views in this article do not necessarily reflect those of the United States Air Force.

1 *Datamation*, March 1, 1971, p. 43.

2 Alexander, T., "Computers Can't Solve Everything," *Fortune*, May 1969.

3 Bratman, H. and Court, T., "The Software Factory," in *Computer Characteristics Quarterly*, Spring 1973.

4 Boehm, B. W., "Software Design," in *Planning Computer Information Utilities*, (eds.) H. Sackman and B. W. Boehm, Princeton Press, 1972.

Answers to A Software Quiz

8

1B. Where Does the Software Effort Go?

	Analysis and Design	Coding and Auditing	Checkout and Test
SAGE	39%	14%	47%
NTDS	30	20	50
GEMINI	36	17	47
SATURN V	37	24	44
OS/360	33	17	50
TRW Survey	44	20	34

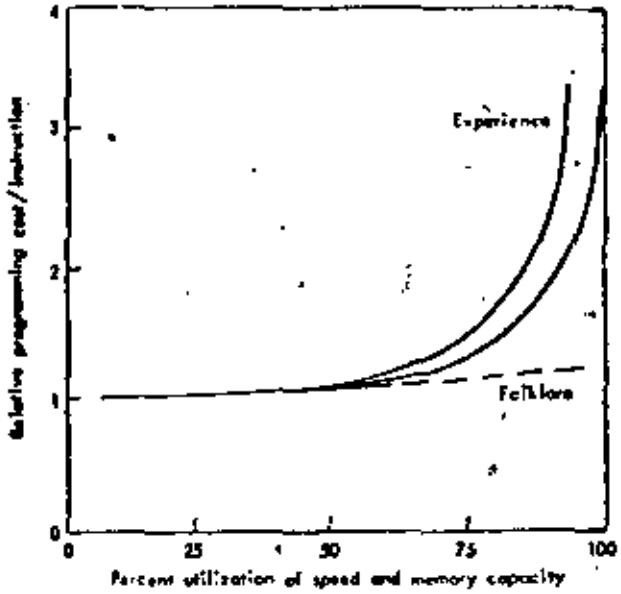
How close did you come to that large 45-50% for checkout? Whatever you estimated, it was probably better than the planning done on one recent multimillion dollar, multiyear (nondefense) software project by a major software contractor which allowed two weeks for acceptance testing and six weeks for operational testing, preceded by a two-man-month test plan effort. Fortunately, this project was scrapped in midstream before the testing inadequacies could show up. But similar schedules have been established for other projects, generally leading to expensive slippages in phasing over to new systems, and prematurely delivered, bug-ridden software.

Another major mismatch appears when you compare the relative amount of effort that goes into the three phases with the relative magnitude of R&D expenditures on techniques to improve effectiveness in each of the phases. Relatively little R&D support has been going toward improving software analysis, design, and validation capabilities.

The difference in the later TRW data probably reflects another insight: that more thorough analysis and design more than pays for itself in reduced testing costs.

(Refs.: Boehm, B.W., "Some Information Processing Implications of Air Force Space Missions: 1970-1980," *Astronautics and Aeronautics*, January 1971. Wolverton, R., *The Cost of Developing Large-Scale Software*, TRW Paper, March 1972.)

2B. How Do Hardware Constraints Affect Software Productivity?



Hopefully, your estimate was closer to the "experience" curve than the "folklore" one. Yet, particularly in hardware procurements, people make decisions as if the folklore curve were true. Typically, after a software job is sized, hardware is procured with only about 15% extra capacity over that determined by the sizing, presenting the software developers with an 85% saturated machine just to begin with. How uneconomic this is will be explained by Fig. 11 in the text.

Those data also make an attractive case for virtual memory systems as ways to reduce software costs by eliminating memory constraints. However, the strength of this case is reduced to the extent that virtual memory system inefficiencies tighten speed constraints.

(Ref.: Williman, A. O., and C. O'Donnell, "Through the Central 'Multiprocessor' Avionics Enters the Computer Era," *Astronautics and Aeronautics*, July 1970.

Software Impact

Increasing software productivity: factors

However, the fact remains that software needs to be constructed, that various factors significantly influence the speed and effectiveness of producing it, and that we have at least some measure of control over these factors. Thus, the more we know about those factors, the more our decisions will lead to improved rather than degraded software productivity. What are the important factors?

One is computer system response time. Studies by Sackman and others[9] of time batch versus on-line programming have shown median improvements of 20% in programming efficiency using on-line systems.

However, in these same studies, *variations between individuals* accounted for differences in productivity of factors up to 26:1. Clearly, selecting the right people provides more leverage than anything else in improving software productivity. But this isn't so easy. Reinstedt[8] and others have shown that none of the selection tests developed so far have an operationally-dependable correlation with programmer performance. Weinberg, in his excellent book,[10] illustrates the complexity of the issue by citing two programmer attributes for each letter of the alphabet (from age and agility through zygosity and zodiacal sign), each of which might be a plausible determinant of programmer performance. Still, the potential payoffs are so large that further work in the areas of personnel selection, training, and evaluation should be closely followed. For example, the Berger Test of Programming Proficiency has proved fairly reliable in assessing the programming capability of experienced programmers.

Other factors such as *programming languages* have made significant differences in software productivity. Rubey's PL/1 study showed differences of up to 2:1 in development time for the same program written in two different languages. In a related effort, Kosy obtained a 3.5:1 productivity improvement over one of the Rubey examples by using ECSS, a special-purpose lan-

[9] ... *Man-Computer Problem Solving*, ... Publishers, Inc., 1970.

[8] Reinstedt, R. N., "Results of a Programmer Performance Prediction Study," *IEEE Trans. Engineering Management*, December 1967, pp. 183-187.

[10] Weinberg, G., *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

Answers to A Software Quiz (cont'd.)

3B. Where Are Software Errors Made?

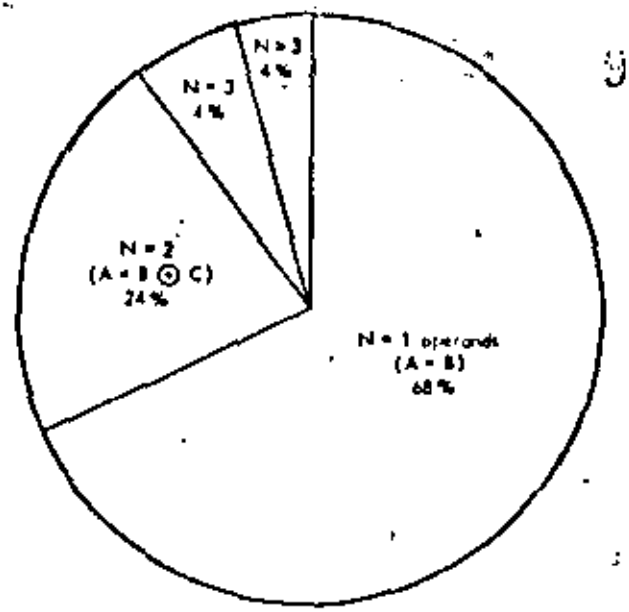
	% errors in each unit (all errors)			
	PL/1	% COBOL % JOVIAL % FORTRAN		
Computation and assignment	9%	23%	26%	20%
Sequencing and control	38	17	27	41%
Input/output	8	8	7	8
Declarations	20	26	20	16
Functional use	21	18	n.a.	n.a.
Compilation dependent	n.a.	n.a.	n.a.	7
Total (%)	100%	100%	100%	100%
Errors (No.)	214	148	243	87

Several points seem fairly clear from the data. One is that GOTO-free programming is not a panacea for software errors, as it will eliminate only some fraction of sequence and control errors. However, as Column 4 shows, the sequence and control errors are the most important ones to eliminate, as they currently tend to persist until the later, more difficult stages of validation on critical real-time programs. Another point is that language features can make a difference, as seen by comparing error sources and totals in PL/1 with the other languages (FORTRAN, COBOL, and JOVIAL), although in this case an additional factor of less programmer familiarity with PL/1 also influences the results.

(Refs.: Rubey, R. J., et al, *Comparative Evaluation of PL/1*, United States Air Force Report, ESD-TR-68-150, April 1968. Rubey, R.J., *Study of Software Quantitative Aspects*, United States Air Force Report, CR-71150-R0R40, October 1971.)

4B. How Do Compilers Spend Their [illegible]?

NUMBER OF OPERANDS IN FORTRAN STATEMENTS
(Knuth study: 440 Lockheed Programs, 250,000 statements)



It's evident from the data that most FORTRAN statements used in practice are quite simple in form. For example, 68% of these 250,000 statements were of the simple form A=B. When Knuth saw this and similar distributions on the dimensionality of arrays (58% unindexed, 30.5% with one index), the length of DO loops (39% with just one statement), and the nesting of DO loops (53.5% of depth 1, 23% of depth 2), here was his reaction:

"The author once found . . . great significance in the fact that a certain complicated method was able to translate the statement

C(I*N+J):=((A+X)*Y) +2.768((L−M)*(−K)) Z

into only 19 machine instructions compared with the 21 instructions obtained by a previously published method. . . . The fact that arithmetic expressions usually have an average length of only two operands, in practice, would have been a great shock to the author at that time."

Thus, evidence indicates that batch compilers generally do very simple things and one should really be optimizing batch compilers to do simple things. This could be similarly the case with compilers and interpreters for on-line systems; however, nobody has collected the data for those.

(Ref.: Knuth, D.E., "An Empirical Study of FORTRAN Programs," *Software Practice and Experience*, Vol. 1, 1971, p. 105.)

guage for simulating computer systems.

Weinberg has also shown[10,11] that the choice of *software development criteria* exerts a significant influence on software productivity. In one set of experiments, programmers were given the same program specification, but were told either (Group P) to finish the job as promptly as possible or (Group E) to produce as efficient a program as possible. The results were that Group E finished the job with an average of over twice as many runs to completion, but with programs running an average of six times faster.

Another important factor is the software *learning curve*. The table in the next column shows the estimated and actual programming effort involved in producing three successive FORTRAN compilers by the same group.[12]

Compiler Effort No.	Man Months	
	Estimated	Actual
1	36	72
2	24	36
3	12	24

Clearly, software estimation accuracy has a learning curve, also.

But other factors in the programming environment make at least as large a contribution on any given project. The most exhaustive quantitative analysis done so far on the factors influencing software development was an SDC study done for the Air Force Electronic Systems Division in 1965,[13] which collected data on nearly 100 factors over 169 software projects and performed extensive statistical analysis on the results. The best fit to the data involved 13 factors, including stability of program design, percent mathematical instructions, number of subprograms, concurrent hardware development, and number of man-steps—but even that estimate had a standard deviation (62 man-months) larger than the mean (40 man-months).

Increasing software productivity: prescriptions

Does all this complexity mean that the prospect of increasing software productivity is hopeless? Not at all. In

[11] Weinberg, G. M., "The Psychology of Improved Programming Performance," *Datamation*, November 1972.

[12] McClure, R. M., "Progress in Performance in Software Production," in *Software Engineering*, 1970 (J. F. Buxton and B. Randell, NATO, January 1969.

[13] Nelson, E. A., *Management Handbook for the Estimation of Computer Programming Costs*, SDC TM-3225, Oct. 31, 1966.

fact, some of the data provide good clues toward avenues of improvement. For example, if you accurately answered question 1 on the Software Quiz, you can see that only 15% of a typical software effort goes into coding. Clearly, then, there is more potential payoff in improving the efficiency of your analysis and validation efforts than in speeding up your coding.

Significant opportunities exist for doing this. The main one comes when each of us as individual programmers becomes aware of where his time is really going, and begins to design, develop and use thoughtful test plans for the software he processes, beginning in the earliest analysis phases. Suppose that by doing so, we could save an average of one man-day per man-month of testing effort. This would save about 2.5% on our total expenditure on software. Gilchrist and Weber[14] estimate that about 360,000 software practitioners are in the U.S.; even at a somewhat conservative total cost of $30,000 per man-year, this is about $10.8 billion annually on software, yielding a testing savings above of about $270 million per year.

Another opportunity lies in the area of programming languages. Except for a few experiments, such as Floyd's "Verifying Compilers," programming languages have been designed for people to express programs with a minimum of redundancy, which tends to facilitate the coding process, but makes the testing phase more difficult. Appropriate additional redundancy in a program language, requiring a programmer to specify such items as allowable limits on variables, inadmissible states and relations between variables, would allow a compiler or operating system to provide much more help in diagnosing programming errors and reducing the time-consuming validation phase. For example, of the 93 errors detected during execution in Rubey's PL/I study, 52 could have been caught during compilation with a validation-oriented programming language containing features such as those above.

Another avenue to reducing the validation effort lies in providing tools and techniques which can get validation done more efficiently and early during the analysis phase. This is the basic approach taken in *structured programming*. This term has been used to describe a variety of on-line programming tool boxes, programming systems, and innovative structurings of the software production effort. An example of the first is the Flexible Guidance Software System, currently being developed for the Air Force Space and Missile Systems Organization. The second is exemplified by the Technische Hogeschool Eindhoven (THE)[15] and automated engineering design (AED) systems, while innovative structuring may be seen in experiments such as the IBM chief programmer team (CPT) effort.[11] Although they are somewhat different, each concept represents an attempt to bring to software production a "top-down" approach and to minimize logical errors and inconsistencies through structural simplification of the development process. In the case of the THE system, this is reinforced by requiring system coding free of discontinuous program control ("go-to free"). In the chief programmer approach, it is accomplished by choosing a single individual to do the majority of actual design and programming and tailoring a support staff around his function and talents.

As yet, none of the systems or concepts described has been rigorously tested. Initial indications are, however, that the structured approach can shorten the software development process significantly, at least for some classes of programs and programmers. In one case, the use of AED reduced the man-effort of a small system from an envisioned six man-months to three man-weeks. A major experiment using the CPT concept for an 83,000-instruction, [illegible] *Times*) cut expected project costs by 50% and reduced development time to 25% of the initial estimate.

The validation statistics on this project were particularly impressive. After only a week's worth of system integration, the software went through five weeks of acceptance testing by *Times* personnel. Only 21 errors were found, all of which were fixed in one day. Since then during over a year's worth of operational experience, only 25 additional errors have been found in the 83,000-instruction package.[16]

At this point, it's still not clear to what extent this remarkable performance was a function of using remarkably skilled programming talent, and to what extent the performance gains could be matched by making a typical programming team into a Chief Programmer Team. Yet the potential gains were so large that further research, experimentation and training in structured programming concepts was one of the top priority recommendations of the CCIP-85 study.

Improving software management

Even though an individual's software productivity is important, the CCIP-85 study found that the problems of software productivity on medium or large projects are largely problems of management: of thorough organization, good contingency planning, thoughtful establishment of measurable project milestones, continuous monitoring on whether the milestones are properly passed, and prompt investigation and corrective action in case they are not. In the software management area, one of the major difficulties

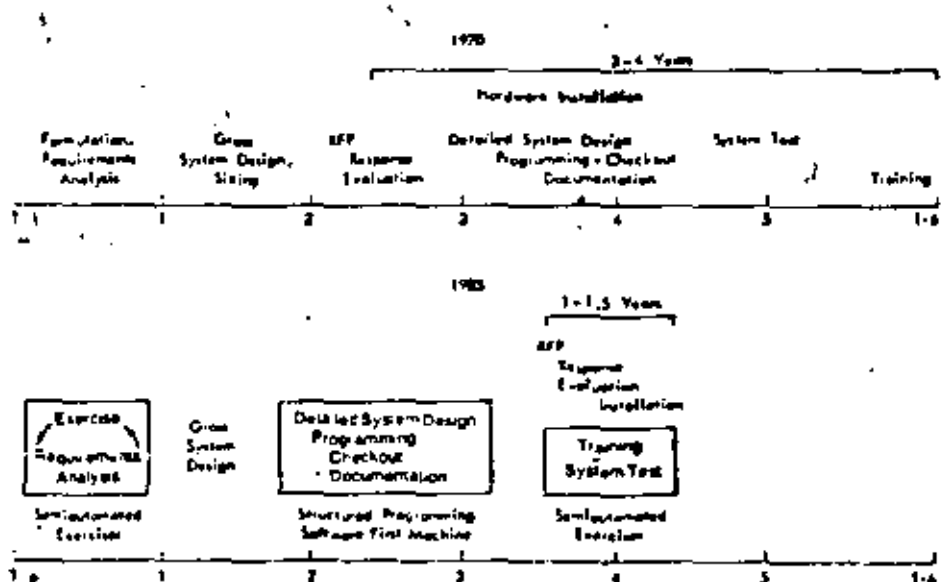


Fig. 9. The software development cycle.

[illegible] B., and K. E. [illegible], "Employment Trained Computer Personnel—A Quantitative Survey," *Proceedings AFIPS 1972 SJCC*, p. 641-646.

[illegible], D. W., *Approaches to Improved Program Validation Through [illegible]*, Memorandum [illegible] Rand Corporation, P-4865, July 1972.

[15] Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System," *ACM Communications*, May 1968.

[16] Baker, F. T., "Chief Programmer Team," *IBM Systems Journal* Vol. 11, No. 1, 1972, pp. 56-73.

[illegible] F. T., "System Quality Through Structured Programming," *Proceedings 1972 [illegible]*, pp. 339-344.

is the transfer of experience from one project to the next. For example, many of the lessons learned as far back as SAGE are often ignored in today's software developments, although they were published over 10 years ago in Mosier's excellent [16] article on the value of milestones, test plans, good interface specifications, integrated measurement capabilities, formatted debugging aids, early prototypes, concurrent system development and performance analysis, etc.[19]

Beyond this, it is difficult to say anything concise about software management that doesn't sound like motherhood. Therefore, this article will simply cite some good references in which the subject is explored in some detail.[20-22]

Getting a far start: the software-first machine

Even if software productivity never gets tremendously efficient, many of the most serious software agonies would be alleviated if we could get software off the critical path within an overall system development. In looking at the current typical history of a large software project (Fig. 9) we can see that the year (or more or more) spent on hardware procurement pushes software farther out onto the critical path, since often the software effort has to wait at least until the hardware source selection is completed.

One of the concepts developed in the CCIP-85 study for getting software more off the critical path was that of a "software-first machine." This is a highly generalized computer, capable of simulating the behavior of a wide range of hardware configurations. Fig. 10 provides a rough plan of such a software-first machine. It would have the capability of configuring and exercising through its microprogrammed control, a range of computers, and could also simultaneously provide some additional hardware aids to developing and testing software.

Suppose a large organization such as the Air Force owned such a machine. The following sequential thing take place: a contractor who is trying to develop software for an airborne computer could start with a need for a machine which is basically the IBM 4Pi, but with a faster memory and different interrupt structures. This software contractor could develop, exercise, store, and recall his software based on the microprogrammed model of the machine. When it turned out that this architecture was hampering the software developers, they could do some hardware/software tradeoffs rather easily by changing the microprogrammed machine representation; and when they were finished or essentially finished with the software development, they would have detailed design specifications for the hardware that could be produced through competitive procurement in industry. Similarly, another contractor could be developing software for interface message processors for communications systems, based on variants of the Honeywell DDP 516; another could be improving a real-time data processing capability based on an upgrade of a CDC 3800 computer on another virtual machine.

The software-first machine could be of considerable value in shortening the time from conception to implementation of an integrated hardware/software system. In the usual procurement process (Fig. 9), the hardware is chosen first, and software development must await delivery of the hardware. With the software-first machine, software development can avoid the wait as hardware procurement is during the system definition phase(s) sary hardware fabrication with start from a detailed design and, with future fabrication technology, should not introduce delays. This saving translates also into increased system operating life, as the hardware installed in the field is based on more up-to-date technology.

However, the software-first machine concept has some potential drawbacks. For example, it might produce a "centrifugal tendency" in hardware development. Allowing designers to tailor hardware to software might result in the proliferation of a variety of similar although critically different computers, each used for a special purpose.

A final question concerning the software-first machine remains moot: Can it be built, at any rate, at a "reasonable" cost? Architectures such as the CDC STAR, ILLIAC IV, and Goodyear STARAN IV would be virtually impossible to accommodate in a single machine. Thus, it is more likely that vari-

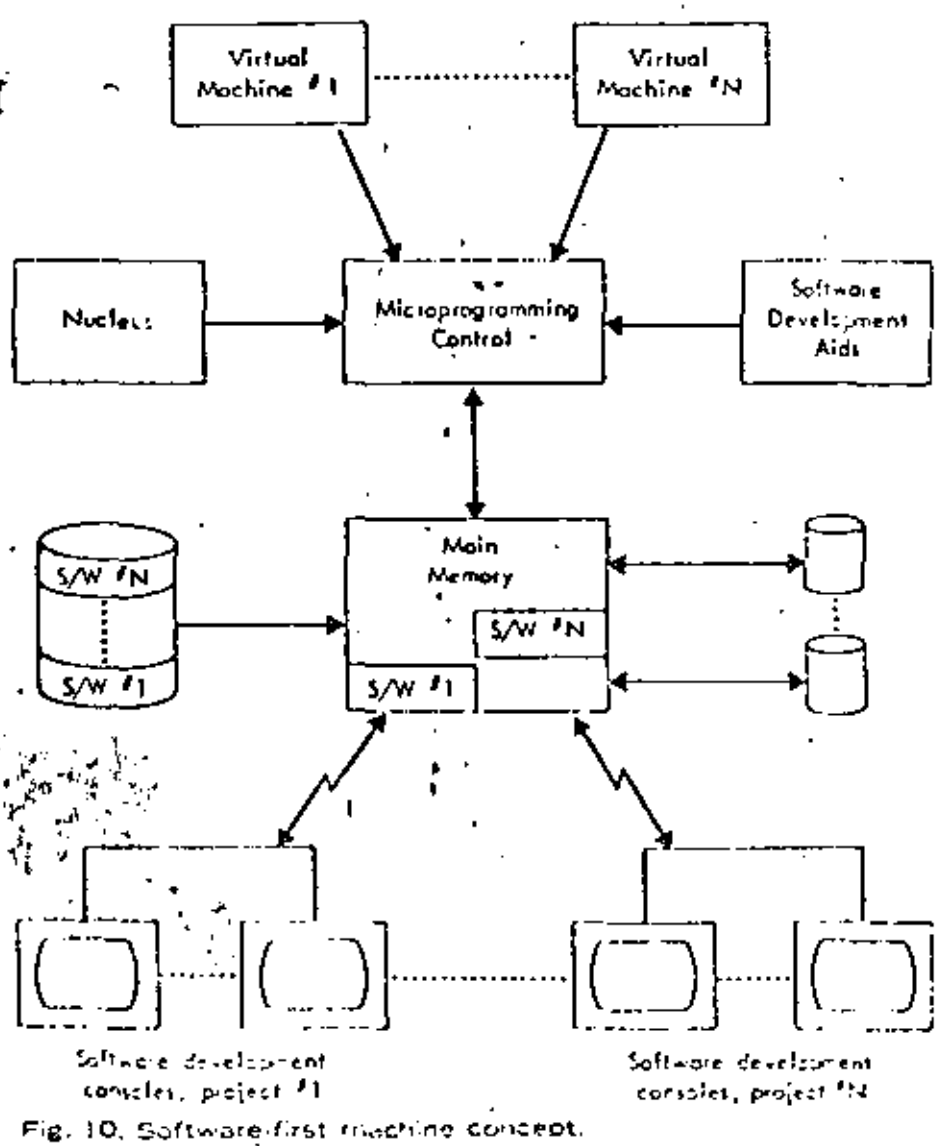


Fig. 10. Software-first machine concept.

[19] Mosier, W. A., "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," IRE Transactions on Engineering Management, Vol. EM-8, June 1961, pp. 99-110.

[20] Naur, P. and B. Randell, eds., Software Engineering, NATO Science Committee, January 1969.

[21] Buxton, J. N. and B. Randell, eds., Software Engineering Techniques, NATO Science Committee, April 1970.

[22] Wegman, G., ed., On the Management of Computer Programming, Auerbach, 1970.

ous subsets of the software-first machine characteristics will be developed for various targets of applications.

One such variant is under way already. One Air Force organization, wishing to upgrade without a simultaneous hardware and software discontinuity, acquired some Meta 4 microprogrammed machines which will originally be installed to emulate the existing second-generation hardware. Once the new hardware is in operation, they will proceed to upgrade the system software using a different microprogrammed base. In this way they can upgrade the system with a considerably reduced risk of system downtime.

Another existing approach is that of the microprogrammed Burroughs B1700, which provides a number of the above characteristics plus capabilities to support "direct" execution of higher-level-language programs.

Other hardware-software tradeoffs

In addition, there are numerous other ways in which cheaper hardware can be traded off to save on more expensive software development costs.

A most significant [illegible] striking difference between "folklore" and "experience" in the hardware-software curves shown in Fig. 2B of the Software Quiz. This tradeoff opportunity involves buying enough hardware capacity to keep away from the steep rise in software costs occurring at about the 85% saturation point of cpu and memory capacity.

Thus, suppose that one has sized a data-processing task and determined that a computer of one-unit capacity (with respect to central processing unit speed and size) is required. Fig. 11 shows how the total data-processing system cost varies with the amount of excess cpu capacity procured for various estimates of the ratio of ideal software-to-hardware costs for the system. ("Ideal software" costs are those that would be incurred without any consideration of straining hardware capacity.) The calculations are based on the previous curve of programming costs and two models of hardware cost: the linear model assumes that cost increases linearly with increases in cpu capacity; the "Grosch's Law" model assumes that cost increases as the square root of cpu capacity. Sharpe's data[23] indicates that most applications fall somewhere between these models.

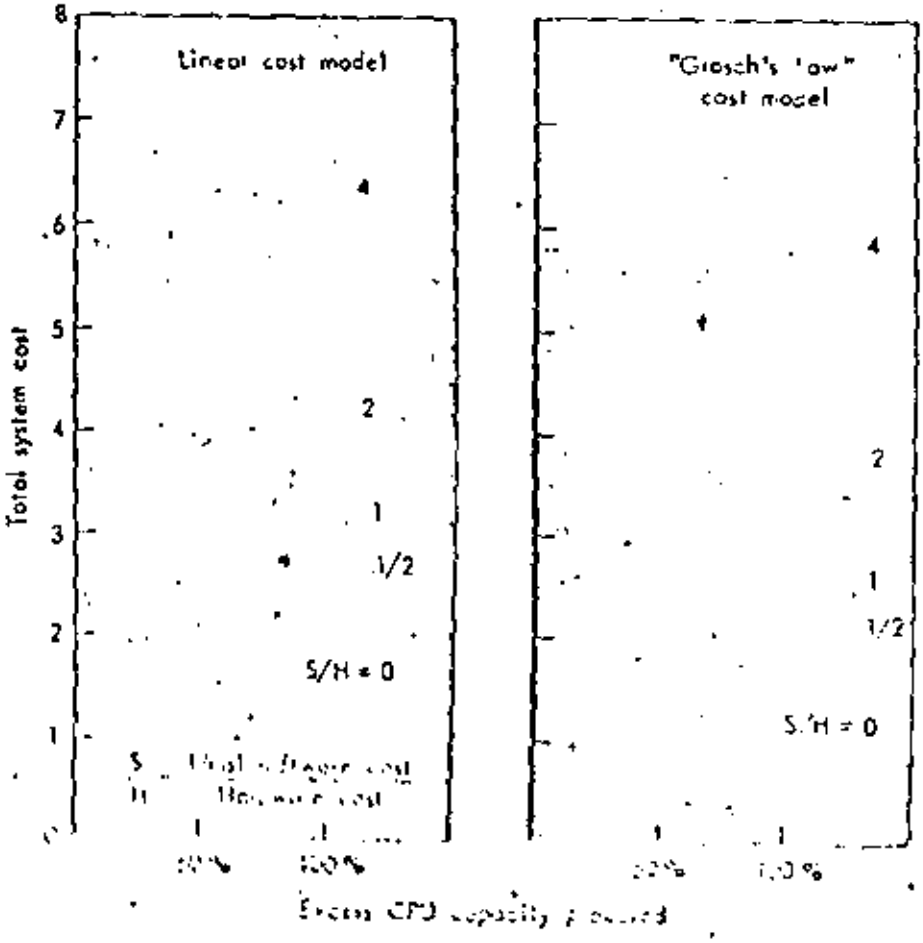


Fig. 11 Hardware-software system costs.

[23] [illegible]

curves are based on [illegible]

they clearly cannot be [illegible] in "[illegible]" fashion by system designers. But even their general trends make the following points quite evident:

1. Overall system cost is generally minimized by procuring computer hardware with at least 50% to 100% more capacity than is absolutely necessary.

2. The more the ratio of software-to-hardware cost increases (as it will markedly during the seventies), the more excess computing capacity one should procure to minimize the total cost.

3. It is far more risky to err by procuring a computer that is too small than one that is too large. This is especially important, since one's initial sizing of the data-processing job often tends to underestimate its magnitude.

Of course, buying extra hardware does not eliminate the need for good software engineering thereafter. Careful configuration control must be maintained to realize properly the benefits of having extra hardware capability, as there are always strong Parkinsonian tendencies to absorb excess capacity with marginally useful tasks.

Software responsiveness

Another difficulty with software is its frequent unresponsiveness to the actual needs of the organization it was developed for. For example, the hospital information system field has several current examples of "wallflower" systems which were developed without adequately consulting and analyzing the information requirements of doctors, nurses, and hospital administrators. After trying to live with these systems for a while, several hospital administrators have reluctantly but firmly phased them out with such comments as, "We know that computers are supposed to be the way to go for the future, but this system just doesn't provide us any help," or, "Usage of the system began at a very low level—and dropped off from there."

The main difficulties stem from a lack of easily transferable procedures to aid in the software requirements analysis process. This process bears an all-too-striking resemblance to the class of folk tales in which a genie comes up to a man and tells him he has three wishes and can ask for anything in the world. Typically, he spends his first two wishes asking for something like a golden castle and a princess, and then when he discovers the operations, maintenance, and compatibility implications of his new acquisitions, he is happy to spend the third wish getting back to where he started.

Similarly, the computer is a sort of genie which says, "I'll give you any

perceived information you want. All you need to do is ask—by writing the software to process it." Or, even, though, we go the one in the last tale one better by canvassing a number of users (or nonusers) and putting their combined wish lists into a software requirements analysis. But our technology for assessing the operations, maintenance, and compatibility implications of the resulting software system is just as inadequate. Thus, large airline reservations software developments (Univac/United, Burroughs/TWA) have reached the point that the customer preferred to wish them out of existence rather than continue them—but only after the investment of tens of millions of dollars. In other cases, where no alternative was available, software rewrites of up to 67% (and in one very large system, 95%) have taken place—*after* the "final" software package had been delivered—in order to meet the user's operational needs.

Considering the major needs for better requirements analysis techniques, the relative lack of available techniques, and the added fact (from Fig. 18 of the Software Quiz) that about 35% of the total software effort goes into analysis and design, it is not surprising that the top-priority R&D recommendation made by the CCIP-85 study was for better techniques for performing and validating information system requirements analyses, and for generating and verifying the resulting information system designs.

The recent Datamation articles on automated system design[1,2] indicated some promising initial developments in this area such as Teichroew's ISDOS project, TOPICS, and IBM's TAG (Time-Automated Grid) system. Other significant aids are being developed in the area of special languages and packages such as SCERT, CASE, CSS, SAM, and ECSS to accelerate the process of design verification by simulating information-processing systems. Also, ARPA's major research effort in automatic programming is focused strongly on automating the analysis and design processes.

Software reliability and certification

Another major area in which the CCIP-85 study identified a serious mismatch between future needs and likely software capabilities was in the area of software certification: of providing guarantees that the software will do what it is supposed to do. (Other significant problem or opportunity areas identified by CCIP-85 included, in order, data security, airborne computing power, multisource data fusion, data communications, source data

mance analysis, parallel processing, and software transferability.)

This is a significant concern right now, but it becomes even more pressing when one extrapolates current trends toward more complex software tasks and toward more and more automated aids to decision making. Just consider the trends implicit in the results of the recent AFIPS Time Surveys[3] which indicated that currently *30% of the labor force must deal with computers in their daily work, but only 15% of the labor force is required to have any understanding of computers.* Extrapolating this trend into the 1980s, as is done in Fig. 12, indicates that perhaps 40% of the labor force will be trusting implicitly in the results produced by computer software.

Software reliability: problem symptoms

Will software be deserving of such trust? Not on its past record. For example, some of the most thoroughly tested software in the world is that of the Apollo manned spaceflight efforts. Yet on Apollo 8, an unforeseen sequence of astronaut actions destroyed the contents of a word in the computer's erasable memory—fortunately, not a critical error in this case. And on Apollo 11, the data flow from the rendezvous radar was not diverted during the critical lunar landing sequence, causing a computer overload that required astronaut Armstrong to divert his attention from the process of landing the spacecraft—fortunately again, without serious consequences. And during the 10-day flight of Apollo 14, there were 18 discrepancies found in the software—again fortunately, without serious consequences.

Other space missions haven't been so fortunate. Recently a software error aboard a French meteorological satellite caused it to "emergency destruct" 72 out of 141 high-altitude weather balloons, instead of interrogating them. An early U.S. Mariner interplanetary mission was lost due to a software error. And the Soviet Union has had missions fail because of software errors.

Down on earth, software reliability isn't any better. Each new release of OS/360 contains roughly 1,000 new software errors. On one large real-time system containing about 2,700,000 instructions and undergoing continuous modifications, an average of one software error per day is discovered. Errors in medical software have caused people to lose their lives. And software errors cause a constant stream of social dislocations due to false arrests, incor-

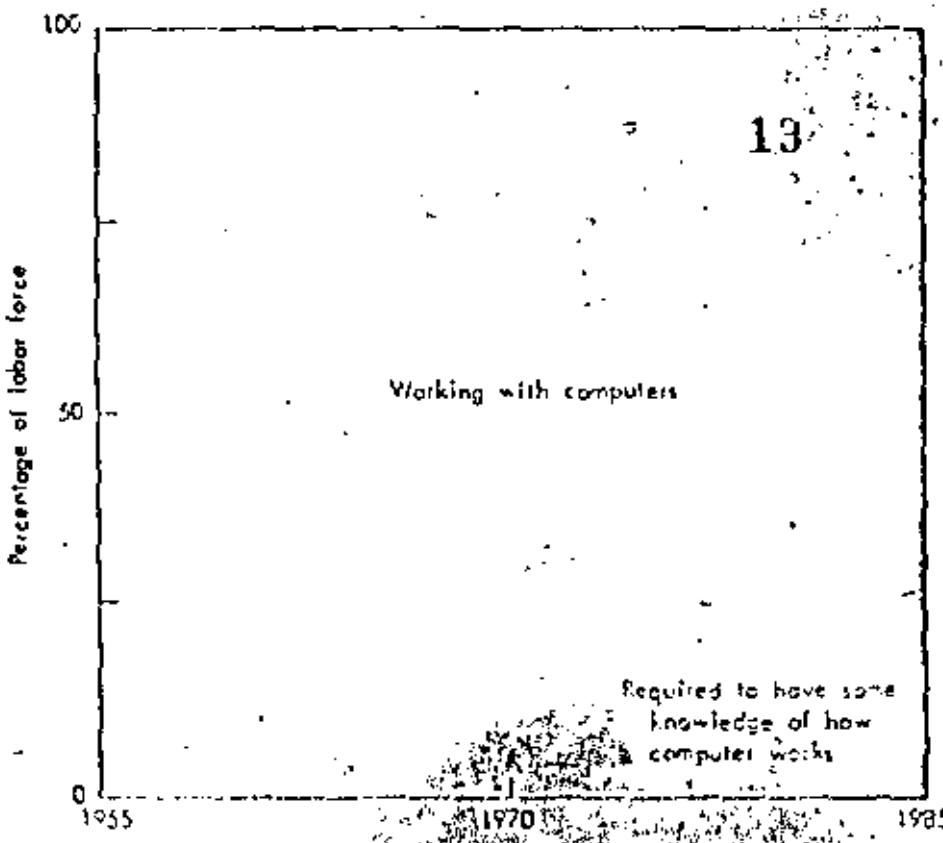


Fig. 12 Growth of trust in computers and software.

rect bank balances or credit records, lost travel reservations, or long-delayed payments to needy families or small businesses. Also, lack of certification capabilities makes it virtually impossible to provide strong guarantees on the security or privacy of sensitive or personal information.

Software reliability: technical problems

As the examples above should indicate, software certification is not easy. Ideally, it means checking all possible logical paths through a program. There may be a great many of these. For example, Fig. 13 shows a rather simple program flowchart. Before looking at the accompanying text, try to estimate how many different possible paths through the flowchart exist.

Even through this simple flowchart, the number of different paths is about ten to the twentieth. If one had a computer that could check out one path per nanosecond (10^{-9} sec), and had started to check out the program at the beginning of the Christian era (1 A.D.), the job would be about half done at the present time.

So how does one certify a complex computer program that has incredibly more possible paths than this simple example? Fortunately, almost all of the probability mass in most programs goes into a relatively small number of paths that can be checked out.

But the unchecked paths still have some probability of occurring. And, furthermore, each time the software is modified, some portion of the tests must be repeated.

Fig. 14 shows that, even for small software modifications, one should not expect error-free performance thereafter. The data indicate that small modifications have a better chance of working successfully than do large ones. However, even after a small modification the chance of a successful first run is, at best, about 50%. In fact, there seems to be a sort of complacency factor operating that makes a successful first run less probable on modifications involving a single statement than on those involving approximately five statements—at least for this sample.

At this point, it's not clear how representative this sample is of other situations. One roughly comparable data point is in Fig. 3B of the Software Quiz, in which only 7% of the errors detected were those made in trying to correct previous errors. The difference in error rates is best explained by both the criticality of the application and the fact that the modifications were being made in a software validation rather than a software maintenance environment.

In another analysis of software error data performed for CCIP-85 by McGonagle,[*] 19% of the errors resulted from "unexpected side effects to changes." Other sources of errors detected over three years of the development cycle of a 24,000-instruction command and control program are shown in Table 1. These data are of particular interest because they provide insights into the *causes* of software errors as well as their variation with type of program.

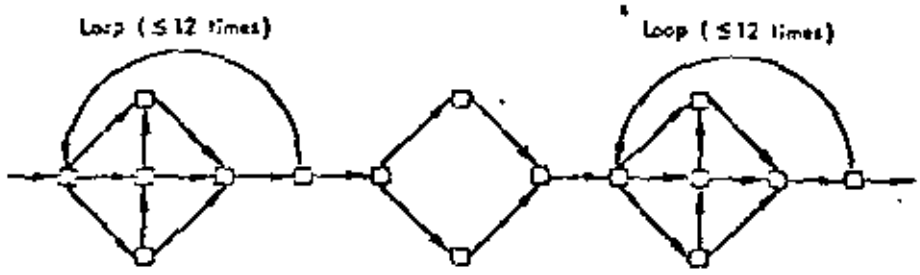


Fig. 13.

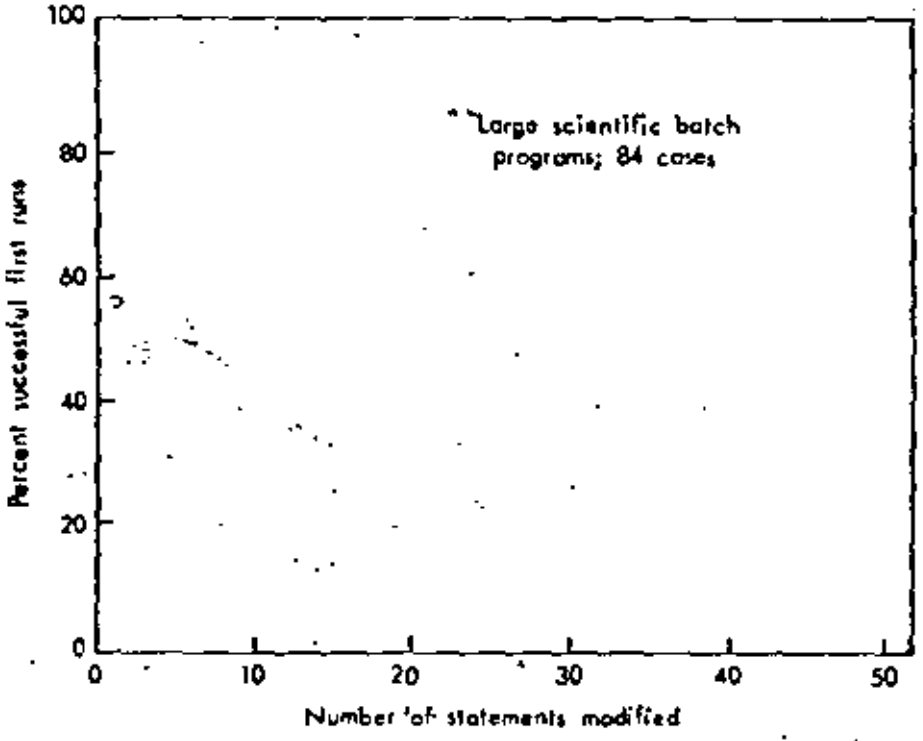


Fig. 14. Reliability of software modifications.

Certification technology

Against the formidable software certification requirements indicated

	Hardware Diagnostics (%)	Executive (%)	User Programs (%)	Total (%)
Unexpected side effects to changes	5	25	10	19
Logical flaws in the design				
Original design	6	10	2	
Changes	9	15	8	1
Inconsistencies between design and implementation	5	30	10	22
Clerical errors	40	20	50	28
Inconsistencies in hardware	40	—	20	11
	100	100	100	100
Total errors detected, 3-year sample	36	108	18	
Number of instructions	4K	10K	10K	

Table 1. Distribution of software error causes.

[*] McGonagle, J. D., *A Study of a Software Development Project*, James P. Anderson and Co., September 21, 1971.

technology [illegible] a great deal [illegible] desired. One organization paid $750,000 to test an 8,000-instruction program, and even then couldn't be guaranteed that the software was perfect, because testing can only determine the *presence* of errors, not their *absence*. The largest program that has been mathematically proved correct was a 433-statement ALGOL program to perform error-bounded arithmetic; the proof required 46 pages of mathematical reasoning.[8]

However, there are several encouraging trends. One is the impressive reduction of errors achieved in the structured programming activities discussed earlier in this article. Another is the potential contribution of appropriately redundant programming languages, also discussed earlier. A third trend is the likely development of significant automated aids to the program-proving process, currently an extremely tedious manual process. Another is the evolutionary development and dissemination of better software test procedures and techniques and the trend toward capitalizing on economies of scale in validating similar software items, as in the DOD COBOL Compiler Validation System. But even with these trends, it will take a great deal of time, effort, and research support to achieve commonly usable solutions to such issues as the time and cost of analytic proof procedures, the level of expertise required to use them, the difficulty of providing a valid program specification to serve as a certification standard, and the extent to which one can get software efficiency and validability in the same package.

Where's the software engineering data base?

One of the major problems the COCOMO-85 study found was the dearth of hard data available on software efforts which would allow us to analyze the nature of software problems, to convince people unfamiliar with software that the problems were significant, or to get clues on how best to improve the situation. Not having such a data base forces us to rely on intuition when making crucial decisions on software, and I expect, for many readers, your success on the Software Quiz was sufficiently poor to convince you that software phenomena often tend to be counterintuitive. Given the magnitude of the risks of basing major software decisions on fallible intuition, and the opportunities for ensuring more responsive software by providing designers with usage data, it is surprising how little effort has gone into en-

[8] Good, D. I., and R. L. London, "Computer Interval Arithmetic: Definition and Proof of Correct Implementation," *ACM Journal*, October 1970, pp. 603-612.

heuristic compilers, optimizing compilers, self-compiling compilers and the like, has there been an R&D effort to develop a *usage-measuring* compiler. Similar usage-measuring tools could be developed for keeping track of error rates and other software phenomena.

One of the reasons progress has been slow is that it's just plain difficult to collect good software data—as we found on three contract efforts to do so for the COCOMO-85 study. These difficulties included:

1. Deciding which of the thousands of possibilities to measure.
2. Establishing standard definitions for "error," "test phase," etc.
3. Establishing what had been the development performance criteria.
4. Assessing subjective inputs such as "degree of difficulty," "programmer expertise," etc.
5. Assessing the accuracy of *post facto* data.
6. Reconciling sets of data collected in differently defined categories.

Clearly, more work on these factors is necessary to insure that future software data collection efforts produce at least roughly comparable results. However, because the data collection problem is difficult doesn't mean we should avoid it. Until we establish a firm data base, the phrase "software engineering" will be largely a contradiction in terms. And the software components of what is now called "computer science" will remain far from Lord Kelvin's standard:

"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of *science*."

But, in closing, I'd like to suggest that people should collect data on their software efforts because it's really in their direct best interest. Currently, the general unavailability of such software data means that whoever first provides system designers with quantitative software characteristics will find that the resulting system design tends to be oriented around his characteristics.

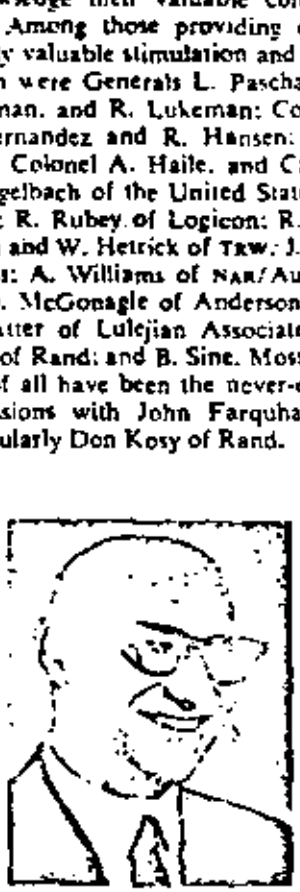
For example, part of the initial design sizing of the ARPA Network was based on two statistical samples of user response, on Rand's JOSS system and on MIT's Project MAC. This was not because these were thought to be particularly representative of future network users; rather, they were simply the only relevant data the ARPA working group could find.

Another example involves the small

completion, several local software designers and managers have expressed a marked interest in the data. Simply having a set of well-defined distributions of program and data module sizes is useful for designers of compilers and operating systems, and chronological distributions of software errors are useful for software management perspective. Knuth's FORTRAN data, excerpted in Fig. 4B of the Software Quiz, have also attracted considerable designer interest.

Thus, if you're among the first to measure and disseminate your own software usage characteristics, you're more likely to get next-generation software that's more responsive to your needs. Also, in the process, there's a good chance that you'll pick up some additional clues which begin to help you produce software better and faster right away.

Acknowledgements. Hundreds of people provided useful inputs to COCOMO-85 and this extension of it; I regret my inability to properly individualize and acknowledge their valuable contributions. Among those providing exceptionally valuable stimulation and information were Generals L. Paschall, K. Chapman, and R. Lukeman; Colonels G. Fernandez and R. Hansen; Lieutenant Colonel A. Haile, and Captain B. Engelbach of the United States Air Force; R. Rubey of Logicon; R. Wolverton and W. Hetrick of TRW; J. Aron of IBM; A. Williams of NAR/Autonetics; D. McGonagle of Anderson, Inc.; R. Hatter of Lulejian Associates; W. Ware of Rand; and B. Sine. Most valuable of all have been the never-ending discussions with John Farquhar and particularly Don Kosy of Rand.



Dr. Boehm is head of the information sciences and mathematics department at the Rand Corp. He has also been a lecturer and instructor at UCLA and was study director of the project discussed in the article. He is a member of many scientific organizations, serves on several government advisory panels, and has published a wide selection of scientific papers. He has a PhD in mathematics from UCLA.

Peter Freeman
University of California, Irvine

It is essential that one have a c eptual under<t nding of a complex activity in order to master its intricacies. Without such a framework, one has only i lated facts and techniques whose interrelationships may be obscured. Without an understanding of broad classes of phenomena, one is condemned to understand e new instance by itself.

Design of complex objects such as large information systems is such an activity. The n r of techniques, tools, pieces of information, and caveats from experience with which one must deal are numerous. The time taken to master the activity of large-scale design is measured in years of full-time activity -- if it can be mastered at all! can be taught to take square roots or to integrate simple equations without knowing a n of the philosophy of mathematics, but it is doubtful that one can acquire even modest skill in system design without having a firm grasp of the philosophy of design.

There is no single philosophy of design, however -- neither now or ever (for the more complex forms of it, at least). What we have is a large number of similar, but differing, views as to the nature of design and as to its basic concepts. As a result, it is impossible to train designers in a short time or even to introduce all of the writings on the philosophy of design.

In this section, we will try to point out very briefly what some others have said about the nature of design and to present our viewpoint in order to clarify some of the presentations of this tutorial. Acquistion of a more complete understanding of the nature of design -- and, indeed, development of an individual viewpoint -- will require additional study and reflection.

VIEWS OF DESIGNING

Designing is one of those interesting human activities that is all around us, in which everyone engages in one form or another, and that eludes precise characterization because of its many forms and its complexity. Design in other fields such as architecture and mechanical engineering is an old and well-established activity. While designing software has its own characteristics, it also shares much with design in other fields. As a prelude to the study of some of the more detailed ideas currently prevalent in the design of software, it is instructive to consider what others have written about the general nature of design.

J. Christopher Jones in his book Design Methods [6] describes a number of the new design methods that have been developed for architectural, urban, and industrial design. Some of these are of quite general applicability (and consequently not very powerful) and may be of use in some software design situations.

In his introduction, Jones pulls out one-line definitions of design from a number of design philosophers (i.e. people who have given serious thought to the nature of design and its underlying, generalizable concepts). These descriptions of designing are quite varied, but do share the common theme of addressing the process of design, not the results:

> "Finding the right physical components of a physical structure."
>
> "Decision making, in the face of uncertainty, with high penalties for error."
>
> "A goal-directed problem-solving activity."
>
> "Simulating what we want to make (or do) before we make (or do) it as many times as may be necessary to feel confident in the final result."
>
> "The conditioning factor for those parts of the product which come into contact with people."
>
> "Engineering design is the use of scientific principles, technical information and imagination in the definition of a mechanical structure, machine or system to perform prespecified functions with the maximum economy and efficiency."
>
> "Relating product with situation to give satisfaction."
>
> "The performing of a very complicated act of faith."
>
> "The optimum solution to the sum of the true needs of a particular set of circumstances."
>
> "The imaginative jump from present facts to future possibilities."
>
> "A creative activity -- it involves bringing into being something new and useful that has not existed previously."

Simon's interesting lectures The Sciences of the Artificial [9] should be read by anyone concerned with design and computing. In them, he states simply that "design is concerned with devising artifacts to attain goals." Included in his lectures are brief descriptions of the areas in which a designer must be knowledgeable and of the fundamental aspects of design. Of special note is his emphasis on the concepts of hierarchy, allocation of the designer's

resources during design, [illegible] rather than optimizing, in complex design situations -- all concepts that should be familiar to the software designer.

One of the standard works on traditional engineering design is the book by Asimov [2]. The opening sentance of the book provides a succinct view: "Engineering design is a purposeful activity directed toward the goal of fulfilling human needs, particularly those which can be met by the technological factors of our culture." The remainder of his book describes a set of steps that can be followed to achieve that objective.

The innovative thinker on architectural design, Christopher Alexander [1], who is often quoted by those concerned with software design, stresses the desired result of design:

> ... every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble into form and context.

Lucas [7], in introducing a new approach to the design of information systems for organizations, focuses on a particular aspect of the design environment which he has deemed to be of paramount importance: "Creative design techniques are focused on solutions to organization behavior problems in systems design." This focus on one part of the problem is typical of most attempts to improve actual design practice and can be seen, for example, in the works of those whose concern is reliability or efficiency or user-centeredness. This is not surprising since, as one goes down to more detail from the philsophical statements quoted above, the drive to be operational forces one to select particular elements of the problem on which to concentrate.

THE PURPOSE OF TECHNICAL DESIGN

Most of the views of design quoted above rightfully emphasized the larger purpose of most design activities -- creating artifacts to attain goals in a social context. The matrix of activities germane to any design activity is quite involved and extensive. One must consider organizational and behavioral factors that lie in the domain of the social sciences, managerial concerns that are addressed by management science, aesthetic concerns (Yes! Even in software!) peculiar to the technology, [illegible]

Our concern here is limited [illegible] technical aspects of design, not because they are necessarily the most important but because without a solid understanding of them, the other concerns have little meaning. If one steps back from the particular design techniques discussed in this volume, three basic purposes of design can be discerned (remember, hereafter we are focusing on the technical domain only):

- discovery of problem structure;
- creation of the outlines (architecture, logical structure) of a solution for the problem;
- review of the results to ascertain if they meet the stated goals.

There is nothing magical or immutable about these objectives of design and, indeed, others might characterize things differently. This is simply one way of abstracting from the details involved in the study of design techniques.

Good designers understand intuitively that a successful design is one that matches the structure of the problem for which the designed object is a solution. What is often difficult to understand for beginning designers is that most design situations are incredibly complex and that the structure of the problem that seems to be obvious at first glance may not accurately reflect of the real situation. The first purpose of design, then, must be to discover enough of the structure of the problem that there is a reasonable chance of devising an appropriate solution.

As an example, consider a small but quickly growing company. As the organization grows, the old manual accounting procedures become too slow and inadequate for meeting the demands of a growing company needing more frequent and current information on its operations. A simple-minded approach would be to simply automate the functions now carried out by hand. But would that be sufficient? Probably not.

Some of the other aspects of the problem that must be considered in this situation include: how to convert easily from existing systems, training of personnel in use of the new system, design of interfaces so that current personnel can use the new system without extensive training, provision of backup and security of critical files, how to provide for expansion of the system in the future as the company continues to grow in ways and amounts that are not predictable because of the nature of the company's business base, the economics of various possible implementations, interfacing with other planned and existing systems of the company (inventory, billing, project management), and so on.

---

*"Satisficing" is a term introduced by Simon to describe the common situation in which we will accept a solution that meets all our constraints, even though it may not be an optimal solution. In most design situations, the complexity makes it difficult, if not impossible, to find optimal solutions.

The point is that there are many visible factors that will impact the eventual success of the system being designed. While one in general cannot consider all possible factors, the most significant must be identified and their influence taken into account. Discovering the structure of the problem means understanding the interactions between these factors and the desired functional characteristics of the system.

Once we think we understand the problem, the next major step is to develop the outlines of the solution. This is the creative aspect of design in the strict sense of the word, although developing an accurate understanding of the problem requires just as much creativity in many cases.

The major activity is the establishment of the architecture of the system. That is, we engage in a combination of spelling out in general terms how the artifact will look to the user -- what functions it will perform -- and how it will be built -- the major algorithms and data representations it will use.

Some parts of this process of spelling out the overall structure may require that we carry the design down to a very detailed level in order to determine the feasibility of performing certain functions. But, in general we are establishing the major pieces of the system, their relationships, interfaces to other systems and the outside world, and carefully specifying what must be done along with rough indications of how it is to be done.

The third purpose of design is to review repeatedly what has been done so far, to compare it to what is desired, and thus to evaluate progress. Review takes place at all stages of the development cycle, of course, but it is most central to the design phase. Review of code production is intended to determine that what has been implemented is what was specified; review of test results is meant to confirm that a sufficient set of tests have been run; review of specifications seeks to determine if the formally stated requirements of the customer have been captured in operational terms. Review at the design phase, though, goes beyond just determining if something that has been previously spelled out has indeed been done -- it is an integral part of the process of discovering the nature of the problem and the proper structure of the solution.

DESIGN METHODS AND METHODOLOGIES

Before we discuss software design in more detail, software design, it will be useful to consider design methods and methodologies. Most readers have probably heard of top-down and bottom-up design methods and may have heard of various design methodologies. It is counterproductive to attempt to define narrowly any of the terms used in design, but consideration of broad definitions at this stage will clarify later comments.

What is a method? Simply, it is a way of doing things -- in this case, designing. A method for finding the roots of a polynomial or for balancing a general ledger is usually well specified with little or no room for alternative courses of action. A method for solving urban social problems will not be so well-defined and leave much (usually too much) room for different paths to be taken by the person following the method. This range of constraint on problem-solving methods -- from precise algorithms on the one hand to loose collections of heuristics on the other -- is true of design methods as well. While precise software design methods can be constructed (for example, steps to follow in designing a symbol table system), the design methods usually considered for medium to large-scale software systems tend toward the heuristic end of the scale.

For our purposes, a design method specifies three things: 1) what decisions are to be made, 2) how to make them, and 3) in what order they should be made. A particular method may ignore or leave implicit one of these elements and in most cases the statement of the method overall is done informally (far from an algorithmic representation)! Let us look briefly at several commonly encountered design methods.

The best-known method is called top-down design. Let us look at the three parameters of this method. The decisions to be made at any point in time are those that affect the greatest possible amount of the total design. The order then follows: if we have grouped decisions into classes or levels, then we make decisions at the highest level first and then iteratively make decisions at the lower levels. Decisions within levels are usually made randomly, or more correctly, according to some other method. The level of a decision is often defined in terms of its distance away from actual implementation concerns. Thus, deciding on the form of a function is high-level while deciding on the format of a symbol table is low-level. Various means of making decisions may be specified, but a general caveat in top-down design is to make decisions that take into account as many as possible of the relevant design goals and constraints and that restrict the set of alternatives for lower-level decisions as little as possible.

A closely related method is called the outside-in method. Basically it is the same as top-down, only the sense of direction is defined in terms of the outside of the system (what the user sees) versus the inside (the implementation). Obviously this may correspond to top-down if one defines top-level decisions as those that deal with the functions of the system. On the other hand, if one defines top-level as those decisions dealing with the overall structure of the implementation, then the two methods will specify different orders of making decisions. In fact, the outside-in approach has been developed largely to counteract the tendency of some designers even when using a top-down approach to pay insufficient attention to the needs of the end users.

The obvious corollary of outside-in design is the inside-out design method. When using this approach, one makes decisions relating to the implementation of the system first before making decisions concerning the external functions of the system. Of course, one must have a clear concept of what is needed on the outside before internal decisions can be made. This method tends to define less closely what decisions are to be made in contrast to the top-down and outside-in methods. The latter broaden the class of decisions to be made to include those that affect the functional architecture of the system. The criteria for making decisions is also more oriented toward satisfying design objectives relating to the implementation of a system. The techniques for making these decisions can thus be more rigorous since one is dealing with time-space tradeoffs which can be quantified as opposed to qualitative measures of user satisfaction (in the case of external functions).

Another method often discussed is the bottom-up approach. In this method, one makes the lowest-level decisions first and gradually builds up the capabilities of the system. The decisions to be made are determined by where one is in the process, starting with decisions concerning basic building blocks and internal functions of the system and proceeding up to decisions concerning external functions. The decision rules used also depend on the point in the process: the low-level decisions tend to concentrate more on internal criteria, of course, the later decisions are more influenced by what is available from the lower levels of design.

A final design method is the most-critical-component-first attack. In this approach, one designs first those parts of the system whose operation is most constrained (e.g. a real-time processing module in a command and control system). The criteria for making decisions is to make them so that the desired critical parameters are satisfied. Once the critical components are designed, decisions can be made according to some other method.

It is important to stress that none of these "definitions" should be taken as hard and fast. Further, rarely are they ever followed in precise fashion. A strict top-down approach usually will only work in examples in which a prior design effort has already determined what is possible at the lower levels. A real design effort using a top-down approach will normally be mixed with bottom-up and/or critical-component-first design approaches. Thus, the descriptions of these methods should be taken as conceptual models, not as rigorous prescriptions.

One often hears the term "design methodology." It is important to understand the distinction between a method and a methodology -- especially since the distinction is not always carefully maintained. In essence, a methodology is a collection of methods, chosen to complement one another, along with rules for applying them. Thus, a comprehensive design methodology would spell out several methods (for example, top-down and critical-component) and provide rules for when to use one and when to switch to another. More generally, design methodology includes management techniques, documentation procedures, tools to aid the designer, standards for specifications that serve as the input to the design process, and standards for the output of the process which must serve as the input to the implementation procedures. A good example of a methodology is the chief-programmer team concept of IBM which specifies management techniques, programming procedures, documentation standards, and so on.

The important point made in this section is that there are different approaches to design -- different methods -- and that isolated methods must be coupled with other methods and rules for applying them in the form of methodologies. We have not discussed how one chooses particular methods nor how one puts together methods to form a methodology. These topics, while extremely important, are beyond the scope of our considerations here.

FUNDAMENTAL DESIGN ACTIVITIES

It is important to remember that the main purpose of this paper is to abstract from the particular techniques of design discussed elsewhere in this volume. The state of understanding of the design activity is not sufficient to permit a complete model or abstraction to be built that will accurately describe in general terms all, or even a significant fraction, of the activities observable in designing. With this warning in mind, let us briefly consider five general operations that can be found in several different places in the design process:

-operationalization;
-abstraction;
-elaboration;
-verification;
-decision-making.

Operationalization

Making something operational means making it usable or functional. It may involve changing non-testable constraints (e.g. "The system should respond quickly") into ones that are. It may mean expressing vague requirements (e.g. "The system should have a comfortable interface") in terms of explicit functions to be performed and data to be represented.

Stating that we want a system to be user-centered is all very well, but this is not operational since it is not clear what is meant by this statement and we cannot unambiguously test for its fulfillment. If we opedationalize it, we will have to list out specifically the characteristics the system must possess in order to be user-centered. Another typical non-operational statement often found in design situations is the requirement that the

system must provide "good" response, without spelling out in testable terms what is meant by "good".

Design ideas and concepts also require operationalization. The purchaser of a system may have an idea of a system to support executive decision making. But until this idea is operationalized by specifying what the system is to do, the idea will be unrealized. Likewise, one may understand the concept of implementing a set of decisions as a finite state automaton, but the concept must be made operational by spelling out what the input parameters, states, and outputs are to be.

Abstraction

Abstraction is the operation of generalizing, of throwing away irrelevant details, of separating the essentials of a situation from the inessential, of considering something as a general quality unrelated to any particular concrete object. We abstract in many situations in order to see the overall structure of something. Theorems in mathematics, physical laws, and generalizations of all kinds are abstractions that permit us to better understand individual situations by relating them to other situations of similar characteristics. Abstraction has long been used in programming through the use of subroutines in which we abstract a set of operations into a single name. The value of this type of abstraction is extremely important at the level of design as well.

Examples of abstraction in design are abundant. When we start to work on a design, we are working with the concepts of the end user, stated in the language of the problem domain. For example, we may need to deal with accounts receivable, billing ledgers, order books, posting, balancing, billing, and so on. While each of these concepts will eventually have a realization in the machine in terms of variables and operations in some programming language or in hardware, they are not tied down at the initial stage -- that is, they are more abstract.

There is another dimension in which abstraction or generalization is valuable. Suppose we are presented with the requirements for a data management system in which there are demands to enter data, change it, list it out for checking, delete it, change its ordering, select subsets of it for printing in any of several different formats, produce backup tapes of the data base, produce output tapes for distribution, and so on. This list of requirements (even in the partial form presented here) is somewhat bewildering. By abstracting and noting that we have editing functions, output functions, and maintenance functions, however, we can get a clearer picture of the problem situation. This in turn will help us to devise an appropriate system design.

Elaboration

Although not a complete complement, elaboration is in many ways the opposite of abstraction. To elaborate something means to make it more detailed, to add features, to work it out to perfection. Top-down design procedures make heavy use of elaboration, but other approaches and aspects of design involve elaboration as well. If we are following a most-critical-component-first design approach, then, after devising the critical components, we elaborate our design by adding additional less-critical parts. When we take a general set of functions and list the particular functions implied, we are elaborating. When we take a gross representation of control flow and add the housekeeping details we are elaborating.

When we take abstractions and convert them into equivalent functions and representations expressed in other terms, it is a form of elaboration, but not one same precisely as adding detail. In this case we are really carrying one set of abstractions into another set. They may be more detailed, but need not be. For example, we may have started our design dealing with abstract data types such as sales orders, then converted these into blocks and lists, then taken these abstractions into arrays and pointers, and finally taken the representation into a form directly implementable in the memory structures of an actual machine. This refinement process is essential to the use of abstractions.

Verification

To verify is to ascertain the correctness of something. Many times in the development process we must compare our developing system to requirements, specifications, standards, and constraints. We verify the performance of a system, we test modules to determine if they do what they are supposed to do, and so on. The process of verfication, not necessarily the technical process of verification, pervades the development process.

Decision-making

In the last analysis, decision-making is what design is all about. We first identify what decision must be made, then generate alternatives among which to decide, evaluate these alternatives to form the basis for a rational decision, and finally choose one alternative. This process is interrelated with and uses all the other fundamental operations of design. As we discussed above, the order and form of decision-making is determined by the method and the relationship of decision-making to other aspects of the methodology.

CRITICAL PARAMETERS

There are three parameters of the design process that are especially critical to its success: the representation used, the experience of the designer, and the information available. We will briefly discuss the nature of each of these parameters and illustrate their importance to design.

Representation

One way of characterizing design is that it is a process of building a representation of the object to be created. We take it as self-evident that some representation of a piece of software exists from inception until a binary image is loaded into memory for execution or an internal form is given to an interpreter.

There are several reasons why design representation is important: Representations, in some form other than contents of a human memory, are needed so that the information involved can be communicated to others; the limited capacity of the human mind for technical detail must be augmented (represenations of current software systems often run to hundreds or even thousands of printed pages); the form of the representation may greatly affect the performance of various design tasks (review, backup, prediction of results); the representation of information affects our success or failure in producing more refined representations that are closer to the desired final result.

A design is a representation of an object. Fundamentally, a design is a collection of information that tells us something about the object we want to create. Thus, the design process can be viewed as an information-gathering process that continually expands the design by putting more and more information into the representation until a termination point is reached. Typically, we terminate a design when we have enough information to implement the object. Figure 1 illustrates this informational view of design.

It follows then that design is a process of deciding what information to include in the design and what information not to include. This has led us in an earlier paper [4] to propose that it is primarily the decisions made in the course of design that must be represented.

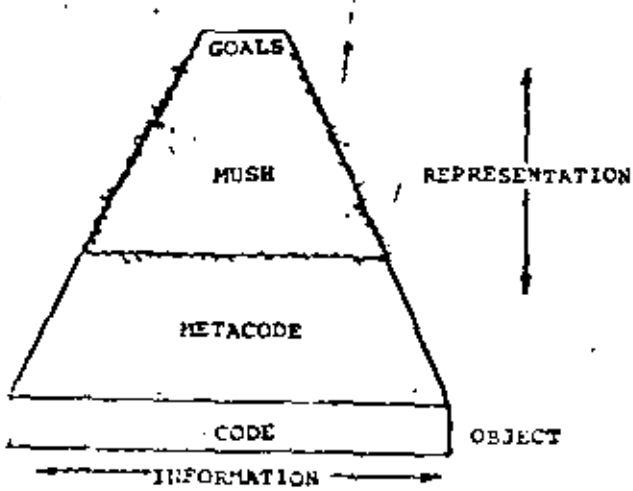


Figure 1: An Informational view of Design

It is important to distinguish representations from documentation. The current use of the word "documentation" usually denotes information describing an object such as a program, an operating procedure, or the results of a design process. It may also connote specifications that "document" a set of needs. This word has the further connotation in many instances, especially in design where the object does not yet exist, of representing everything we know about the object.

It is precisely this connotative use of the word "documentation," however, that we believe has prevented clear thinking about design representation. In the case of a program that exists, it is understood that the documentation augments the actual object and that we must go to the object itself as the final authority on questions of substance. (This in fact is a standard maxim given to those attempting to understand a program). In the case of a program design, however, the "documentation" is all that exists. Thus, it is more than information about an object (i.e., documentation); it is a representation of the object being created.

The failure to be clear about what the flowcharts, lists, diagrams, and voluminous prose generated during design really constitute has prevented us from thinking clearly about the role of these representational forms. Of course, everyone recognizes that the stacks of paper generated during a design process contain (in theory) all the information we have available about the object (usually, there is additional, unrecorded information). Yet, few people have specifically treated such ad hoc collections of information as a representation in the same way that an architect views graphical representations of a building or an engineer views mathematical representations of a physical process. Just as other design professions have carefully evolved (and in many cases consciously created) the representational forms they use, so must we do the same for software design.

Recently there has been increased concern taken with software design. Some of this work has explicitly addressed the issue of representation. Structure charts [10] and program description languages [3,11] are both forms for representing program designs and have been used with some success. Peters and Tripp [8] survey a number of graphical design representation schemes.

Our own work on software design rationalizations* [4] provided another form for representing a design; while typically requiring too much effort to carry out in the full format proposed earlier, its ability to capture the decisions being made during design led to one of the representational forms discussed in [5].

---

* Software design rationalization is a technique for capturing not only the design decisions, but the reasoning that led to them. Specifically, one is instructed for each decision, to list a set of alternative

Knowledge

The role of knowledge in design is fairly obvious. Since the objective of design is to make rational choices among alternatives in order to fulfill a set of requirements, there is a need for information on which to base decisions. We need knowledge of two types: pre-existing and developed.

Pre-existing knowledge includes information about the equipment and other systems to which the new systems must interface, knowledge of the operational requirements of the system, and knowledge of alternative structures. For example, in the design of a text-processing system, the designer should know about alternative ways of representing text, of searching for occurrences of strings, and so on.

Software designers do not have handbooks which parallel those available to designers in other fields. Even if there existed, for example, information on a variety of software structures, there would still be a need for other important classes of information. During design, especially if it deals with a new situation, many needs for information arise. In the design of a terminal interface language, for example, we may need to know the preference of potential users for alternative forms of commands. Or, in the design of a symbol table, we may need to know the expected usage density for the particular situation at hand. This type of information must be developed by small experiments or analyses carried out during design.

There is a limit to the amount of information that can be used effectively in the design process, however. If one has too much information to process in making a design decision, the effort expended in evaluating it may exceed the gains from making a slightly more rational decision. Evaluating this type of tradeoff requires experience.

Experience

The experience that a designer must have, just as the information he or she should possess, is of several types: experience at designing, with similar design situations, and with the technologies that must interface to the design process.

As we have noted, designing is a complex process that cannot be captured in the form of a set of explicit instructions. One must learn how to design by doing it. But, beyond this elementary stage, many of the operations of designing require a large amount of intuition and decision-making that cannot be expressed entirely as rational decision. For example, knowing how many alternatives to consider, which to look at first, which to reject out of hand, and so on, are decisions that generally are made well only if one has made them before and learned from the experience. Allocating the resources of the design team requires experience with what types of activity will pay off best. Since such decisions come up repeatedly and in many different forms, one must be prepared to make them quickly.

Experience with previous designs of a similar nature is critical. Many decisions cannot be made entirely rationally either because of lack of hard information on the alternatives or because the interactions between a large number of factors cannot be clearly discerned. Further, knowledge of similar designs provides one with a prototype on which to base the current design. This type of experience is more than just simple knowledge, since often the prototype serves more as a guide to decision-making than as a template.

The third type of experience that is essential is with the technologies to which design interfaces. If a designer has no personal knowledge of programming, it will be difficult for him or her to specify feasible structures that can be easily programmed. Knowledge of the limits and possibilities of hardware are essential to the software designer. Experience with creating new system organizations can be a great aid. Experience with managing both the design process and the implementation procedures again provides the designer necessary insights to some of the factors that may influence the eventual success of the design.

DESIGN AS A GOAL-DRIVEN ACTIVITY

It is clear that design is an activity with a definite goal in view - creation of an artifact that meets certain goals. The obvious goal in most design situations is to create a system that provides certain functions. All too often in software design, this seems to be the only goal to which much attention is paid.

There are other very important goals for many systems, however, such as reliability, user-centeredness, maintainability, efficiency, and security. Increasingly, these other goals are seen to be at least as important as the goal of providing certain functions.

It is a fundamental characteristic of designing that it permits us the opportunity to achieve any set of desired goals. If we want a system to be very user-centered, we have the opportunity to design it that way (or, at least to emphasize that goal within other existing constraints). On the other hand, if we do not pay attention to all of our goals during design, there is usually little that can be done later to incorporate them into the system. If we have devised an artifact without paying attention to whether

---

choices, an evaluation for each alternative, and the reason for the one actually chosen. While it is clear that this provides a great amount of information not normally found in a design, which may be very useful to later attempts to understand the design, it is not clear that the added effort of producing this detailed "rationalization" is cost-effective. The decision statement representation discussed below appears to be a more natural representation for actual design situations.

it possesses certain qualities or not, then it should be no surprise that it doesn't have them!

The primary way in which objectives enter into the design process is their use in making decisions. If we are seeking a highly reliable system, then each design decision should be evaluated as to whether it will have a significant impact on the system reliability or not. If it does, then we can make the decision on the basis of which alternative contributes most to reliability. In most design situations we have a number of objectives which must be balanced. Not all will be equally applicable to every decision, but for those that do apply to a given decision we must consider their interactions and relative importance if they interact in a negative way.

Finally, all of the major objectives of a design effort should serve as guidelines for choice of the overall system structure. As we have pointed out above, a successful design should fit the structure of the problem situation. One of the determining factors of the problem environment is precisely the conditions we place on possible solutions -- i.e., the objectives of the design.

CONCLUSION

This short paper certainly has not covered all there is to say about the area of design -- that was not it's purpose. It has presented enough of the conceptual aspects of design, however, to indicate the types of concerns that go beyond the simple presentation of design techniques. Studying the philosophy of design may not translate immediately into improved design performance, nor may it ever be directly identifiable as a precise cause of improved performance. But, as with most activities, a deeper understanding of the design process will eventually pay off. It is for this reason that we have introduced you to this aspect of software design.

REFERENCES

1. Alexander, Christopher. Notes on the Synthesis of Form. Harvard University Press, 1964.

2. Asimov, Morris. Introduction to Design. Prentice-Hall, 1962.

3. Caine, Stephen H. and E. Kent Gordon. "PDL - A Tool for Software Design," Proc. AFIPS 1975 NCC.

4. Freeman, Peter. "Toward Improved Review of Software Designs," Proc. AFIPS 1975 NCC.

5. Freeman, Peter. "Software Design Representation: Analysis and Improvements," TR81, ICS Department, Univ. of California, Irvine, 1976.

6. Jones, J. Christopher. Design Methods. WileyInterscience, 1970.

7. Lucas, H. Toward Creative Systems Design. Columbia University Press, 1974.

8. Peters, L.J. and L.T. Tripp. "Design Representation Schemes," Proc. 1976 NAI Symposium, IEEE Press.

9. Simon, H.A. The Sciences of the Artificial. MIT Press, 1969.

10. Stevens, W.P., G.J. Myers, and L.L. Constantine. "Structured Design," IBM Systems Journal 13,2 (1974).

11. Van Leer, P. "Top-Down Development Using a Program Design Language," IBM Systems Journal 15,2 (1976).

IMPROVEMENTS IN THE SYSTEM LIFE CYCLE

Daniel TEICHROEW

Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, Michigan, USA

24

(INVITED PAPER)

Reprinted with permission from *Proc. IFIP Congress 1974*
North-Holland Publishing Company

The paper surveys developments and trends in information processing system life cycles. Awareness of the life cycle is increasing, and recognition of the diversity of desirable system properties is focusing attention on the tradeoffs which must be made in design. The data base subsystem is receiving increased attention. Emphasis is shifting from the subsystem activities (such as programming) to system-level architecture and design. These trends apply also to the information processing activities of system professionals (analysts, designers, programmers, etc.) and will gradually lead to an integrated, computer-aided, data-base-oriented information processing system designed for them.

1. INTRODUCTION

The number of computer based information systems has increased rapidly in the past two decades. The size and capability of individual systems has also increased. Many professionals engaged in analysis, design, implementation and operation of systems are not satisfied with the progress that has been made - systems take too long to build, they cost much more than predicted and do not work as promised when installed. There are many differing opinions on what are the underlying problems, what are symptoms rather than problems, and what are the most promising avenues of improvement. For example, one recent expression of the difference is shown in the reply by Larsen (1973) to an article by Boehm (1973).

This paper is concerned with the engineering of systems to serve the information needs of organizations. The major task is the structuring of its subsystems (applications software, data base, computer systems and non-computerized procedures). In order to accomplish the design it is frequently necessary to design, i.e., structure, each of these subsystems to several lower levels. Several major trends in system engineering (as applied to information processing) are examined in the following sections. The first trend (Section 2) deals with recognition of the life cycle process. Section 3 is concerned with the growing awareness of the interrelationship (frequently conflicting) of a number of desirable properties of systems. An aspect that is receiving growing attention is the data base subsystem (Section 4). Section 5 is concerned with the formalization of the design process.

All of these trends point to improvements in methodology for information systems analysis, design, implementation, operation, and maintenance. Problem solving and decision making methodology are being applied to system engineering. In addition, system engineering is also being applied to the system engineering process itself. The progress in this direction is outlined in Sections 6 and 7.

2. INFORMATION PROCESSING SYSTEM LIFE CYCLES

The basic steps in the life cycle of information systems (initiation, analysis, design, construction, test, installation, operation, and termination) appeared in the earliest applications of computers to organizational problems. [See for example Schrimpf and Compton (1969), Aiken (1954), Canning (1956), and Gregory and Van Horn (1960).] The need for formal, exhaustive procedures for carrying out the life cycle was recognized; early examples are the IBM (1961) SDP publications, the Philips ARDI method (1968), and the SDC method, Willworth (1964). In the last few years, a large number of books and papers on this subject has been published.* A number of generalized systems are commercially available, and most organizations have either adopted one of these or developed their own. In the majority of cases the latter procedure has been followed since it appears that the systems department of an organization will not accept a procedure unless it has been involved in the development of the procedure.

These procedures manuals normally include: (1) a set of activities which must be carried out, frequently presented as a PERT diagram, (2) a set of forms to be completed at various stages of the development, which comprise "documentation," and (3) a project management system, sometimes including a computerized recording and reporting system to measure progress against the plan. These procedure manuals have grown out of practical experience and have received little formal analysis. One exception is the work of Belady and Lehman (1971), referenced by Ramey (1973), in developing a macro-model applicable to large programming system projects. They refer to an earlier paper, Lehman (1969) which describes an integrated family of tools for a total system development methodology, particularly directed to limiting the propagation of errors in successive releases of software systems. These models are qualitative and useful for understanding of the process, rather than providing or improving specific methods.

Considering the growing importance of information systems, and the tendency for projects to increase in size and importance, it is strange that more effort has not been devoted to this area. Probably the major reason is that everyone has been fully occupied in improving the individual subactivities of the process where more could be accomplished in a reasonable period of time. Furthermore, without extensive education and training there is difficulty in, and resistance to, using tools developed by others even in the same organizations. The growing recognition of the interrelationship of the activities can be expected to lead to improvements outlined in Sections 6 and 7.

3. SYSTEM PROPERTIES

Perhaps the most striking trend in information systems, at least as represented by the literature is the change in relative importance of certain

*The life cycle model is an important element in the information system curricula proposed by the ACM Curriculum Committee Computer Education for Management; Ashenhurst (1972), [illegible]er (1973).

computer science community appeared concerned primarily with whether an "algorithm" worked, whether it was stated in an acceptable language, and how elegant it was. The implementors of systems were primarily concerned with whether the system could fit into the available memory space and time constraint, and secondarily with the cost of development and maintenance.

In the last few years the importance of certain other properties has become evident both in the literature and to implementors. Among the most important is the recognition of the need to be able to "maintain" the system and to "adapt" it to changes. An early example of this change appears in the reports of the NATO Conferences. In the first report [Naur and Randell (1969)], the criteria mentioned are reliability and logical completeness. The second report [Buxton and Randell (1970)] is concerned with quality (correctness of the product in performing the specified task and its performance efficiency) and flexibility (which includes portability and adaptability).

Boare (1972) gives a list of 18 properties, and summarizes the current situation as: "The first necessity is for the software designer and his customers to recognize that there is a conflict of objectives, and that there is the need for compromise." Myers (1973) is an example of the growing trend by proponents of new methods to indicate their effect on the various properties. Another definition of some properties is given by Waters (1972).

Unfortunately there is as yet no generally accepted set of definitions for these properties. A proposed classification is outlined in fig. 1. The first level of classification divides properties into three classes: functional - what the system does; performance - how well (quantitatively) it performs a function; other properties - which can only be described qualitatively.

Each of these classes can be subdivided further. Functional capabilities consist of two subclasses: capabilities (operations which the system can perform), and interfaces and constraints (interconnections which the system is able to accept). The performance characteristics can be divided into those dealing with the system itself, and those dealing with the process by which the system is developed. The "other" characteristics are divided into (qualitative) properties of the system in operation (such as reliability and recoverability), and those dealing with its ability to be modified or changed to suit changes in the system environment (such as flexibility, adaptability, and maintainability).

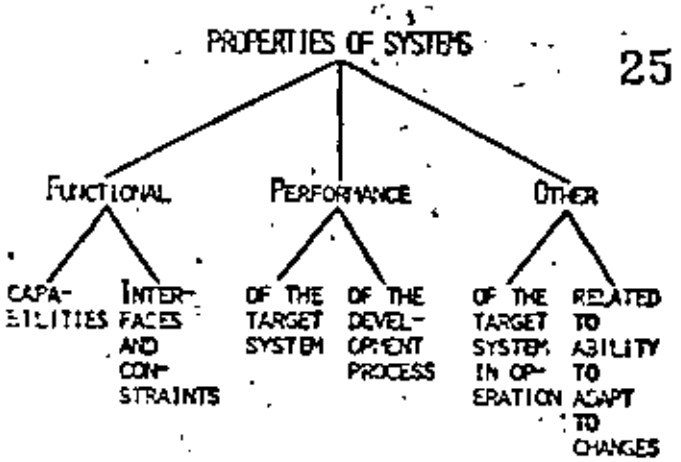


Fig. 1. Taxonomy for properties of systems.

4. IMPORTANCE OF THE DATA BASE SUBSYSTEM

Until recently, the order of importance of components of information processing systems was considered to be: hardware, application software, data base, and finally, non-computerized procedures. This was certainly true in scientifically oriented installations where some very large and powerful computers had simple paper tape inputs. Computer science literature has emphasized algorithms and paid little attention to data until recently.* The problems associated with data have been well known in the business data processing field, but even there have received less attention than other aspects, e.g., hardware.

One indication of change is the new emphasis on the data base in the design of a system. Arising out of the recognition that if a system is to have a long life with a minimum of changes, the system design should concentrate on those parts of the system that are the most stable. The parts that change frequently should be designed as easily modifiable additions. In many cases, the data base structure is the most constant; the processing and reports required from the system change more rapidly. Therefore, the data base should be designed first, in such a way that it need not be changed as the procedures or reports are changed. This concept is now being followed, at least implicitly, in the design of many large systems. [See for example Zwanevelt (1973).]

A natural consequence of the data base orientation in design is the development of software which provides standard procedure-independent methods of structuring and accessing. In recent years such software, under the name of data base management systems (DBMS) has received a great deal of attention by user-oriented organizations such as CODASYL, software firms and equipment manufacturers, though until very recently it has received almost no treatment in the computer science literature. DBMS's are currently going through a learning curve; the considerable effort devoted to them is decreasing excessive overhead, and they are becoming an essential part of most medium and large installations. There is every indication that in the next generation of systems software, the data management facilities will be integrated into, and play a major role in, operating systems. (Since data base management systems can make programs more data independent, the life of both programs and bases can be extended. This forms a strong motivation for systems departments to organize their activities around a DBMS despite the cost and reorientation required.)

The need to standardize data elements at various levels - application, organization, nationally and internationally - is also being recognized.+ Standardization is proceeding under the auspices of the American National Standards Institute (ANSI, 1973). Interest is growing in procedures and software such as data dictionary and data directory programs to manage data element names [Uhrowitz (1973)]. The data element management system is one example of development of information system for use by analysts and programmers.

* See for example the statement of L.D. Yarborough (1973) in his review of a survey of programming languages: "We venture to predict significant change in programming, with primary emphasis on data description, the expression of algorithms taking a distant second place. In a sense this must happen, as far too many data errors plague our current use of algorithms."

+ Several hundred people recently attended the Symposium on the Management of Data Elements in Information Processing held by the National Bureau of Standards in Washington, January 1974.

5. SYSTEM DESIGN AND ARCHITECTURE

The increasing awareness of the fact that tradeoffs have to be made among desirable properties of systems (since improving one property frequently has adverse impact on some other properties) has resulted in the recognition of the need for an overview of the whole system and its structure similar to that provided by architects. The term "architecture" is being applied to information systems, though not yet consistently.*

Among recent papers discussing the application of architectural concepts to information systems are Scarrott (1972) and Spiro (1973). The architecture approach shifts attention from individual subsystems (operating systems, applications software, data base management systems, etc.) to the structure of the system as a whole. The need for this is increasingly being recognized as, for example, by Baer (1971) in a review: "...modular programming is not sufficient: modular architecture would be more relevant." [See also Pyle (1972) and Spooner (1971).]

Once the relative importance of the different properties is determined and the overall structure of a system selected, there remain alternative ways of proceeding. Several aspects of these alternatives are discussed in the following subsections. One important dimension is how the activities are structured in relation to the major levels in the hierarchy, e.g., top-down, bottom-up, and how sharply the total process should be divided into sub-activities. Another dimension is concerned with how the desirable properties are to be achieved, i.e., by following guidelines, successive approximations, or by selecting "optimum" values.

5.1 Structuring of sub-activities

There is as yet no agreement on how sharply the distinction between analysis, design and implementation should be made and, if it is made, whether the structuring method should be the same in each case.

Gregory and van Horn (1960) for example distinguish between analysis and design and characterize analysis as "top-down" and design as "bottom-up": "The designer must synthesize or put together the parts to devise an operating system. This is an entirely different job from that of the analyst, who analyzes - literally, takes apart - the process he is investigating in order to understand it."

Ogdin (1972) argues against a separation of design and implementation: "The distinction between design and implementation is unclear. It is not possible to specify a system and then turn the system over to a programmer for separate implementation. There must be a continuing dialog between the eventual user, the system designer, and the programmer. This is one reason that programmers' utility programs are usually so well implemented: the eventual user, the system designer and the programmer are all the same person!"

He then describes current practice in program design and implementation as "oscillating between the top-down and bottom-up techniques." One of the strong advocates for the top-down approach is Baker (1972). In his already classic paper on the "Chief Programmer Team" approach he advocates it for both design and programming.

The bottom-up approach to [illegible] stability a complete statement of the requirements at the [illegible] detailed level. The design process then [illegible] grouping the elementary units of requirements into larger units. These are then grouped into still larger units. This eventually leads to subsystems and then to the target or object system. An example of the approach is that given by Langefors (1973) and an implementation in a computer program by Briggs (1966). It is the basis of the approach of Nummaker (1971) mentioned below.

One of the major difficulties with the bottom-up approach is the level of detail to which individual data elements, processes, etc. have to be defined before they can be combined. It is therefore reasonable to try to develop larger "blocks" of which information systems are composed which might then be assembled or synthesized to form the system. In order to make this method efficient, it is necessary to identify a relatively small number of blocks, or types of blocks, which the developer has to understand and remember. In order to maintain generality, it is then necessary to provide for parameters so that the blocks can be adapted to individual situation. This approach is called the "synthesis" or "functional" approach. An example of the development of "user" functions is given by Welsh (1969). A recent attempt to expand the functional approach to design is given by Becker (1972) and Booth (1973). Becker describes the structure of information network by six levels, of which level 1 is the information network. At Level 2 two functions are separated: network processing and information processing. Both subdivides information processing into five levels and presents the functional approach as a step towards simplifying the design process itself as well as simplifying the subsequent choice of obtaining or creating hardware and software elements. He describes the functions in sufficient detail so that the designer can synthesize his design by selecting the functions he needs.

Myers (1973a, 1973b) advocates top-down software design (which he calls Composite Design) and then discusses the relative advantages and disadvantages of top-down and bottom-up "coding." He concludes: "I am somewhat biased toward bottom-up development... [however] the arguments for and against top-down development and bottom-up development aren't convincing. Either one can be used to develop a program designed with Composite Design. However, it is important to choose one or another and to stick with it."

Whether logical design should be top-down or bottom-up and how much the logical system design should be separated from the physical system design depends on the status and requirements of the particular information system. Life cycle methodology should be adaptable to any combination.

5.2 Performance

Some discussions of the growing importance attached to properties related to the modifiability of systems imply that performance is now of secondary importance. For example, Donaldson (1973) states: "Recent shifts in emphasis have occurred in the field of software development. The primary requirement to be met in software development has always been to perform the function specified for the software. But, where at one time secondary emphasis was placed only on software efficiency, that is, core and time required, today three other factors are recognized as requiring special emphasis. These factors are reliability, maintainability, and extensibility."

*For example, Brooks (1971) in a review states: "I would dispute that the paper describes an architecture, a term originally introduced and generally used to mean a functional description of a structure, as visible to a user. The author used the word in a quite different sense, and his definition of it isn't clear."

[illegible] (1972) gives a vivid picture of the result of ignoring performance: "A newly-designed airplane, for example, usually either flies or crashes into the nearest hillside. A newly designed on-line computer system, on the other hand, often behaves in a manner analagous to an airplane taking from New York to Los Angeles." His book, which is intended to be a text on the design of on-line computer systems, [illegible] the problem of designing performance into systems by the following statement: "To do a complete job, we should also be able to perform a cost-effectiveness analysis of our system, a subject which is beyond the reach of this book!...".

In a similar vein Booth (1973), in his book describing the functional approach, limits his coverage by the following statement (pp. 18-19): "The functional approach does not, of course, solve all of the designer's problems. It does not deal with the quantitative or performance factors which are necessary to completion of design...".

While no one would argue that available cost benefit analysis and design tools are perfect, there are some available. It is generally accepted that a program cannot be made reliable by testing; reliability must be built into it. A similar question arises regarding performance - can performance be incorporated into a system after it is built or does it have to be designed in? Clearly any design methodology should incorporate such methods as are available to achieve acceptable levels for performance (as well as for other desirable properties).

5.3 Design approaches

The approaches mentioned below have been selected because they attempt to describe a procedure which can be followed in a specific instance and are intended to be representative rather than exhaustive. They are divided into three broad categories by the [illegible] ree to which the methods provide specific [illegible] dance. The first class consists of rules of thumb, guidelines, principles; here only general guidelines are given. Next are methods which suggest successive approximations; these approaches consist of developing initial approximations and then modifying them until they are satisfactory. Finally are the normative methods which are intended to lead directly to the "best" or "optimum" system.

(i) Rules of thumb, guidelines and principles

A first step in developing procedures leading to "good" systems is to identify and formulate principles or guidelines which should be followed. An example of such a list of principles is given by Chugani (1972) for the design of management information systems. A number of current textbooks treat design at this level. For example, Yourdon (1972) gives the following rules that an applications programmer must keep in mind to achieve efficient programs: "programs must be written in a modular fashion; there must be a good overlap of I/O and computation; file accesses must be minimized."

Such rules of thumb serve as a check list. However, they do not in themselves represent a procedure that a designer can follow, and they may be misleading. For example, Waters (1972) presents some of the limitations of rules of thumb for file design.

(ii) Successive approximation

The next step beyond a set of guidelines is to develop more or less formal procedures for the process. Each of such procedures has appeared in the literature as part of [illegible] the system life cycle [illegible] manuals mentioned in Section 2. These [illegible] usually consist of a sequence of steps [illegible] in narrative form, sometimes supplemented [illegible] charts.

[illegible] of these methods have been adapted to system engineering from methods that were originally developed for general design problems, for example, the IDEALS approach of Medler (1972). He suggests beginning by determining the function that the system is to achieve, within the least restrictive conditions. He then develops several Feasible Ideal Systems (FIS's) from which the Feasible Ideal System Target (FIST) is selected, and then modifies the FIST to handle all "unavoidable" conditions.

Ross (1970), quoting a 1960 MIT report, discusses the outside in versus the inside-out approach. "...successive stages of problem statements are greater and greater refinements of the original statement of the problem. Each stage is represented in and materialized by the language and computer structure. The end result is a sufficiently refined solution to the original goal achieved by a sequence of elaborations, modifications, tests, and evaluations all of which taken together constitute the evolution of an ever-clearer idea of just what the problem is that is to be solved...". Zurcher and Randell (1968) call the approach "iterative multi-level modelling:" "...a method of modelling a computer system design as it evolves, so that evaluation can be made an integral part of the design process."

Dijkstra (1968) in his celebrated paper on the development of THE multiprogramming system describes the process in terms of a system hierarchy. The approach is also mentioned in the reports of the NATO Conference [Naur and Randell (1969) and Buxton and Randell (1970)]. Wirth (1971) calls the process "program development by stepwise refinement ...Programming is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures."

Intuitively this approach, whatever it is called, is appealing and one looks expectantly for successful applications. Unfortunately, experience with the method appears to be limited, at least experience that has resulted in publication. Zurcher and Randell in their paper mention an experimental implementation. Two other attempts have been reported: Graham (1973) describes the development of DES (Design and Evaluation System), and Aslanian (1972) describes a software development project in which evaluation and modelling were integral parts. To the best of our knowledge, none of these experiments has resulted in operational software or in a procedure which can be applied. The approach also has been challenged; Brooks (1972) in his review remarks: "It challenges, carefully and for cause, some widely cherished theories, including Dijkstra's onion-skin design concept."

Nevertheless, the approach has promise and should be pursued further. Problem areas that need attention include a more precise definition of the hierarchy of levels or "levels of abstraction." How this is to be done in any particular case is not clear, despite the inclination by some to dismiss it as obvious.* The second problem is concerned with making the approach work when the individuals who will design and construct the system are not the ones specifying the requirements. A third problem arises from the size of the system; the method may

*For example, the review by Fletcher (1969) of a paper on levels in a file design states: "If you want to know what is implied by filing system modularity, then read this paper; if you already know, proceed to the next review." This implies that file modularity is easy, obvious and well-known, a contention that some might dispute. One intriguing possibility for a formal approach to structuring hierarchies is the algorithm developed by Alexander (1964).

work when the system is small, has to be augmented if the system is large and documentation must be maintained over long periods of time.

(iii) Normative approaches

the successive approximation methods, design decisions are improved by a trial and error process.

normative approaches attempt to formalize the design process as a (multi-level) model in which each sub-model can be used directly to make a good or hopefully "best" selection of the feasible values of the decision variables.

An example of this approach has been developed by Emery (1971). He assumes specifications for the target system are set in terms of "quality" of the output. Dimensions of this quality might be content, accuracy, response time, and so on. These specifications have an impact on the cost of the system and on the value of the outputs of the target system to the users. Curves for the relationships between the specifications in terms of quality and cost can be postulated. The specifications are selected as the values of quality for which the value cost difference is largest.

A more detailed and explicit procedure is described by Stigler (1969) for the design of real time systems. Another example of this approach is given by Nunamaker (1971). He defines explicitly the objective function, the decision variables and the alternatives. The selection of an "optimum" system is then accomplished by an application of an iterative approach in which solution methods such as mathematical programming are used for low level decisions (e.g., determining blocking factors) and heuristic methods in cases where optimization algorithms are not available.

The difficulty with the normative methods is that, in practice, the problem is so large and complex and the number of alternatives so immense and unstructured that it is very difficult to combine them into one comprehensive model. Consequently, there is a tendency to attack individual subcomponents first.* Progress can be made by solving submodels, but since they are part of a larger problem there must be some provision for coordinating the submodels as indicated in the next section.

*A good example is the area of data base design, which is a formidable problem in its own right. See for example Cardenas (1973).

LIFE CYCLE

A number of different techniques, as distinguished from general approaches discussed above are available for use in various stages of design, construction, and operation of information processing systems. Which ones are used depends on many factors, such as the size of the organization, the experience and qualifications of the systems personnel, the function and type of systems, and so on. A broad classification is sufficient to identify some of the ways in which the techniques could be improved. Table 1 lists a number of techniques classified by the phases in the life cycle and by the degree of automation.*

The techniques classified in the first row as "present methods" are representative of practice in the typical medium or large systems department. These techniques range from no formal procedures (in evaluating proposed systems, testing, and maintenance and modification) to manual methods (in design and programming) to computer-aided procedures (in compilation and in operating systems). The major characteristic of this level of evolution is that most of the effort in system design, construction, testing and maintenance is manual.

The second row shows some improved techniques in use in some organizations. The improvements are mainly in the direction of formalizing manual procedures. Some computer-aided stand-alone methods are shown in the third row. Many of the techniques listed in this row are not new; however, their acceptance has not been rapid,+ and it is instructive to consider some of the reasons.

*The techniques listed (which are illustrative rather than exhaustive) are those used in life cycles of general purpose systems. In particular, specialized packages designed to accomplish specific applications such as numerical analysis or payroll are not included.

+For example, the January 1971 issue of DATA PROCESSING DIGEST poses the conundrum: "What do decision tables and ALGOL have in common?" and gives the answer, "both are more honored than used." Timmreck (1973) in his survey of computer selection methodology, mentions the general purpose system simulators, SCERT, CASE, and SAM, and then states, "Canning in a 1966 article suggested that simulation may be the preferred method of evaluation. Experience to date has not supported this contention...."

Table 1

Techniques used in system life cycle classified by activities and degree of automation

LEVEL OF TECHNIQUE	I PERCEPTION OF NEED	II LOGICAL SYSTEM DESIGN	III PHYSICAL SYSTEM DESIGN	IV CONSTRUCTION	V TEST AND CONVERSION	VI OPERATION	VII MODIFICATION & MAINTENANCE
Present Methods	Not formally recognized	Narrative, tables, charts	Manual benchmarks	Flowcharts, programming languages	Ad hoc	Operating systems	No formal procedure
Improved Methods	Capital investment review procedure	Documentation standards	Standards for system design	Programming standards; modular, decision tables	System test standards Emulators	Manual scheduling	Change control procedure
Computer Aids (stand alone)	Investment analysis programs	Problem statement languages	Simulators	Flow charters COBOL aids	Test generators	Computer-aided scheduling	Monitors Computer-aided analysis
				DBMS			

used is that many system departments view any outside software with suspicion. Sometimes this is the result of (unfortunate) experience with trying to implement software produced somewhere else; sometimes it is caused by the NIH ("Not Invented Here") factor. A second reason is that when a formal evaluation is made, the considerations are frequently narrow. For ple, decision table processors may be evaluated team of programmers. Implicitly or explicitly they may use as their criterion reduction in programming time. On this basis alone, decision table processors may not appear effective; though if the criterion were broadened to include the analysis time and maintenance time, the decision table approach might as well be adopted. Third, even in a comprehensive evaluation, these techniques frequently suffer from the fact that they are stand-alone. Their use requires the transformation of input to a form acceptable to the package, and the package produces output which has to be transformed for the next stage. Consequently, the cost of using the package is greatly increased, and the packages are viewed by the intended users as another hurdle in their work rather than as an aid. A fourth reason arises from the fact that packages are often proprietary, hence, attempts are made to hide the exact algorithms or procedures used. This makes it extremely difficult to obtain confidence in the effectiveness of the package.

This listing of difficulties with use of present packages indicates several avenues of improvement. System department managers need to improve their evaluation procedures, particularly in the ground rules by which such studies are carried out, to be sure that the objective function covers the organization and the systems department and is not restricted to only one aspect of the systems department, such as programming. Secondly, the integration of the packages needs to be improved, so the output of one is acceptable as the input to next one in the sequence. This is admittedly a difficult objective to achieve when the packages are produced by different companies, but pressure from users should make it obvious to the software houses that there is an advantage to standardization. A much more difficult but continually important area needing considerably more attention from the computer profession is the resolution of the issue of proprietary software, particularly the elimination of trade secrets as the only way in which investments can be protected.

7. CONCLUSIONS

Continuation of the several trends related to the life cycle of information processing systems have been discussed in Sections 2 to 5, and the evolution of techniques used in the various stages of the life cycle outlined in Section 6 should yield progress in providing techniques that qualify for inclusion in additional rows of table 1. A fourth row might be used to list integrated computer aids and eventually a fifth row for automated integrated systems. This conclusion follows from the observation that the life cycle process is itself an information processing activity and the general trends therefore apply also to this particular system.

ystems departments in the past have been more concerned with bringing the benefits of computer-based systems to their clients than with improving their own operations. They have tolerated manual methods which they would regard as intolerable in their clients' operations. This is now changing.

This will encourage the development of techniques directly relevant to the improvement of performance of the system department. System professionals are recognizing the diversity of desirable properties of these systems. The result will be progress in the direction of a computer-aided process in which data is stored in a computerized data base and made available to those "users" who need it. The users - analysts, designers, programmers, operators, performance auditors, etc. - will work within a structured framework based on methods such as those outlined in Section 5. The software associated with the system will provide initially clerical aids, then computational and data processing aids, and eventually "optimum" results.

Such a system will be data base oriented. As the object system is being developed, all information about it is stored in the data base which will replace manual documentation. The activities in the life cycle can then be structured to make use of whatever data about the object system is already available and to add new data as it is developed. This concept is not new, see for example Trapnell (1969). It is, however, a logical extension of the trends outlined above and eventually will bring to the systems professionals many of the benefits of computer-based systems that they have provided to others but not utilized themselves.

REFERENCES

[1] Aiken, T., Initiating an electronics program, Proceedings of seventh annual meeting, Systems and Procedures Association, 1954.

[2] Alexander, C., Notes on the synthesis of form, Harvard University Press, 1964, 216 pp.

[3] American National Standards Institute (ANSI), Guide for the development, implementation and maintenance of standards for the representation of computer process data elements, X3L5l Task Group (Data Standardization Criteria) of the X3 Sectional Committee, Computers and Information Processing, 1973.

[4] Ashenhurst, R. L. ed., Curriculum recommendations for graduate professional programs in information systems, Communications of the ACM, vol. 15, no. 5, May 1972, 363-398; Computing reviews, vol. 13, no. 10, October 1972, 22,869.

[5] Aslanian, R., Production and evaluation of software systems by modelling, ACM European Computing Symposium, Venice, April 1972, 121-128.

[6] Baer, J. L., Review of ben-Aaron, M., Programming systems standardization - a rationale, Proceedings of Fourth Australian Computer Conference, vol. 1, 309-312 in Computing reviews, October 1971, 22,030.

[7] Baker, F. T., Chief programming team management of production programming, IBM systems journal, vol. 11, no. 1, 1972, 56-73.

[8] Becker, H. B., Information network design can be simplified step by step, Computer decisions, October 1972, 14-17.

[9] Belady, L. A., and M. M. Lehman, Programming system dynamics, or the meta-dynamics of systems in maintenance and growth, Research Report RC3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, September 1971.

[10] Boehm, B. W., Software and its impact: a quantitative assessment, Datamation, May 1973, 48-59.

[11] Booth, Grayce M., Functional analysis of information processing: a structured approach for simplifying systems design, Wiley Interscience Publications, 1973.

[12] Briggs, R. B., A mathematical model for the design of information management systems, MS

of Pittsburgh,)
[13] Brooks, F. P., Jr., Review of Zell, C. G., [illegible], Computer structures: readings and examples, McGraw Hill Co., York, 1971, 668 pp. in Computing reviews, May 1971, 21,279, 245.
[14] Buxton, J. N., and B. Randell eds., Software engineering techniques, Report of a NATO Conference, Rome, Italy, October 1969. (Published April 1970, available from Scientific Affairs Division, NATO, Brussels 39, Belgium.)
[15] Canning, R. G., Electronic data processing for business and industry, John Wiley & Sons, Inc., New York, 1956.
[16] Cardenas, A. F., Evaluation and selection of file organization - a model and system, Communications of the ACM, vol. 16, no. 9, September 1973, 540-548; Computing reviews, 26,117.
[17] Chugani, P., Some design principles for MIS, Data processing, May-June 1972, 172-173.
[18] Couger, J. D. ed., Curriculum recommendations for undergraduate programs in information systems, Communications of the ACM, vol. 16, no. 12, December 1973, 727-749.
[19] Dijkstra, E. W., The structure of the 'THE' - multiprogramming system, Communications of the ACM, vol. 11, no. 5, May 1968, 341-346.
[20] Donaldson, J. R., Structured programming, Datamation, December 1973, 52-54.
[21] Emery, J. C., Cost/benefit analysis of information systems, SMIS workshop report no. 1, Society of Management Information Systems, 221 North LaSalle, Chicago, Illinois 60601, 1971, 48 pp.
[22] Fletcher, J. G., Review of Madnick, S. E., A modular approach to file system design, in Computing reviews, November 1969, 17,842.
[23] Glans, T. B., B. Grad, D. Holstein, W. E. Meyers, and R. N. Schmidt, Management systems, Holt, Rinehart and Winston, Inc., 1968, 340 pp. (Based on IBM's 'Study Organization Plan, 1961.)
[24] Graham, R. M., G. J. Clancy, Jr., and D. B. DeTaney, A software design and evaluation system, Communications of the ACM, vol. 16, no. 2, February 1973, 110-116; Computing reviews, 25,424.
[25] Gregory, R. H., and [illegible] L., VanHorn, Automatic data-processing systems, Wadsworth Publishing [illegible] 1960.
[26] Hanf[illegible], F. M., Modular construction analysis - a tool for scheduling software debugging activities, Fall joint computer conference, 1972, 173-179.
[27] Hartman, W., [illegible] Matthes, and [illegible] Programs, Management Information systems handbook, (ARDI), McGraw Hill, 1968.
[28] Hoare, C. A. R., The quality of software, guest editorial, Software - - practice and experience, vol. 2, 1972, 103-105.
[29] Langefors, B., Theoretical analysis of information systems, Auerbach Publishers, Inc., Philadelphia, 1973, 489 pp.
[30] Larsen, G. H., Software: man in the middle, Datamation, November 1973, 61-66.
[31] Lehman, M. M., The programming process, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December 1969.
[32] Madnick, S. E., and J. W. J. Alsop II, A modular approach to file system design, Spring joint computer conference, 1969, 1-13; Computing reviews, 17,844.
[33] Myers, G. J., Composite design: the design of modular programs, IBM Poughkeepsie Report TR 00.2406, January 1973, 80 pp.
[34] Myers, G. J., Characteristics of composite design, Datamation, September 1973, 100-102. (See also "Letters" in Datamation, December 1973, 162-163.)
[35] Nadler, G., J. Johnson, and J. Bailey, Design concepts for information systems, CAIS monograph, 25 Technology Park/Atlanta, Norcross, Georgia 30071, October 1972, 98 pp.
[36] Naur, P., B. Randell eds., Software engineering, Report of a NATO Conference, Garmisch, Germany, October 1968. (Published January 1969, available from Scientific Affairs Division, NATO, Brussels 39, Belgium.)
[37] Nunamaker, J. F., Jr., A methodology for the design and optimization of information processing systems, Proceedings of the [illegible] spring joint computer conference, AFIPS New Jersey, vol. 38, May 1971, 283-294.
[38] Ogdin, J. L., Designing Reliable software, Datamation, July 1972, 71-78.
[39] Ogdin, J. L., Improving software reliability, Datamation, January 1971, 49-52.
[40] Philips, Information systems handbook (ARDI), Centrex Publishing Co., P.O. Box 76, Eindhoven, The Netherlands, 1968, 600 pp. (Reprinted as Hartman, et al.)
[41] Pyle, I. C., Some techniques in multi-computer system software design, Software - practice and experience, vol. 2, 1972, 43-54.
[42] Ross, D. T., Uniform referents: an essential property for a software engineering language, in J. Tou ed., Software engineering, vol. 1, Academic Press, 1970.
[43] Scarrott, G. G., Computer architecture, Computer bulletin, vol. 16, no. 11, November 1972, 525-527; Computing reviews, 25,269.
[44] Schrimpf, H. W., and C. W. Compton, The first business feasibility study in the computer field, Computers and automation, January 1969.
[45] Spiro, E., The evolution of systems (information systems architecture), Data management, December 1973, 10-12.
[46] Spooner, C. R., A software architecture for the 70's: part I - the general approach, Software - practice and experience, vol. 1, 1971, 5-37; Computing reviews, 12,297.
[47] Stimler, S., Real-time data-processing systems, McGraw-Hill Book Co., New York, 1969, 259 pp.; Computing reviews, 17,391.
[48] Timmreck, E. M., Computer selection methodology, Computing surveys, vol. 5, no. 4, December 1973, 199-222.
[49] Trapnell, F. M., A systematic approach to the development of system programs, Spring joint computer conference, 1969, 411-418.
[50] Unroxrik, P. F., Data dictionary/directories, IBM systems journal, vol. 12, no. 4, 1973, 332-350.
[51] Water, S. J., File design fallacies, The computer journal, vol. 15, no. 1, 1972, 1-4.
[52] Welsh, W. A., Engineered design of EDP systems, Presented at the Systems and Procedures Association, International Meeting, St. Louis, Missouri, October 1968, 15 pp.
[53] Willworth, N. E. ed., System programming management, TM 1578/000/00, System Development Corporation, March 13, 1964 (draft).
[54] Wirth, Program development by stepwise refinement, Communications of the ACM, vol. 14, no. 4, April 1971, 221-227.
[55] Yarbrough, L. D., Review of Cheatham, T. E., Jr., The recent evolution of programming languages, Information processing 71, C. V. Freiman ed., North-Holland, Amsterdam, The Netherlands, 1972, 298-313, in computing reviews, July 1973, 25,353, 320.
[56] Yourdon, E., Design of on-line computer systems, Prentice Hall, New Jersey, 1972, 608 pp.; Computing surveys, 25,040.
[57] Zurcher, F. W., and B. Randell, Iterative multi-level modelling - a methodology for computer system design, Proceedings of the IFIP Congress 68, D.138-D.142.
[58] Zwaneveld, M. J. M., Data base management conceptual overview, Proceedings of European Diebold Research Program Conference, 1973, 37-49.

ON THE MEANING OF DISCIPLINE IN SOFTWARE DESIGN AND DEVELOPMENT

Anthony I. Wasserman
Medical Information Science
University of California, San Francisco
San Francisco, CA 94143 USA

31

Reprinted from *Software Engineering Techniques*, D Hates (ed.), Infotech International (1977). © Anthony I. Wasserman.

KEYWORDS

Software engineering, programming methodology, software design, systematic program development

ABSTRACT

Discipline in software design and development consists of a set of practices intended to improve both the process of software production and the quality of the resulting software. This paper examines the meaning of discipline in more detail, illustrating a number of techniques that should be used to assure the quality of the finished product and outlining some of the areas where additional progress must be made before the construction of software may truly become an engineering activity. A well-organized, deliberate approach to software construction is seen to involve stages of requirements analysis, complete specification, detailed design, implementation, and quality assurance, supported by formal and informal documentation.

THE NATURE OF DISCIPLINE

Discipline, which may be defined as behavior or order maintained by training and control, takes many forms. Discipline for Olympic athletes, for example, involves a re[illegible]n including extensive conditioning, [illegible]ition, and a controlled diet. The mental discipline followed by a chess master involves study and memorization of many games, along with in-depth analysis of board positions and possible moves and countermoves.

A different form of discipline can be seen in engineering fields, depending upon conformance to a set of rules and procedures. An engineering effort involves a specific sequence of steps beginning with a general plan, proceeding through detailed design, and concluding with implementation of the design. At each stage of the process, it is essential to carry out a series of activities in order to guarantee the success of the effort.

In constructing a building, for example, one must begin with artistic renderings, proceed to more detailed architectural design, and finally to blueprints prior to the beginning of construction. Also, the building site must be tested and prepared, checking for things such as drainage characteristics, soil quality, and stability. The various aspects of a building -- the plumbing, the electrical system, the heating and ventilation, and the structural integrity -- must all be designed in harmony in order to assure that the finished building will be of the desired quality and will function properly. Furthermore, all of these design steps precede construction.

The steps involved in the design and construction of buildings are well understood; it is possible to impose a specific methodology on the entire process. At the same time, however, there is adequate room for individual creativity and innovation, since detailed design and blueprints can serve to validate the ideas.

DISCIPLINE IN SOFTWARE ENGINEERING

The premise that an engineering type of discipline should be applied to the process of software design and development resulted in the coining of the term "software engineering" in 1968 [1]. It was recognized that software creation was a hit-and-miss business, based largely on *ad hoc* techniques. A large number of major software projects had failed completely or had been subject to severe time delays, cost overruns, or other serious problems. Many of the software systems exhibited unpredictable, hard-to-correct errors, were poorly documented, and fell short of user requirements in such measures as response time. In addition, these systems were often difficult to maintain, since the code was incomprehensible as a result of poor programming practices.

It became painfully apparent that there was a need to improve the quality and dependability of software production. The techniques being used did not work consistently and had a disturbing tendency to fail whenever a new or complex problem was attempted. The development of suitable *programming methodologies* presented a number of potential benefits, including the following:

1) improved reliability

2) verifiability, at least in an informal sense

3) adaptability of programs

4) comprehensibility

5) effective management controls

6) higher user satisfaction.

Now, as in 1968, however, software engi ring remains more of a wish than a r lity. In the intervening years, however, there has been a great deal of investigation into the nature of programming and the pr ss of software development. The result of this effort has been the recognition that a form of engineering discipline can be applied to software construction with siderable success.

As with the construction of buildings, the construction of software can be separated into a design stage and a construction stage. If we are given a complete and consistent design for a piece of software (software blueprints), then the process of constructing a program which implements that design is generally straightforward.

One of the problems that has recurred consistently in the software creation process is that the design stage is not separated r the construction stage. Instead, there exists an "urge to write code", to begin the construction process before fully determining the nature of the object to be built. While this technique works up to a point, particularly with smaller programs, it has a tendency to collapse with more complex systems. It can be seen, by analogy, that a building contractor, can safely purchase roofing materials, wiring, plumbing fixtures, i lation, and electrical appliances for a house before the design is complete, but is li y to make some incorrect and irrevocable decisions in trying to assemble those components prior to the completion of the design.

Just as the builder must wait for the architect to complete the detailed plans for a building, so must the programmer wait for the "software architect" to complete the system design. This is a key principle in the achievement of a disciplined approach to software design and development.

In the remainder of this paper, a number of approaches and techniques which constitute a disciplined approach to software development will be examined, along with some of the additional steps needed before the goals of software engineering can be achieved.

Understanding the problem

The first step in the software creation process is to gain a complete understanding of the problem and a clear definition of the program requirements. Without this understanding, it is impossible to proceed successfully. In some cases the definition of the problem is straightforward, as in "Print out the first 1000 prime numbers." More often, however, the problem is less clearly defined and not entirely understood, as in "Develop a computer system to help in reservi g and renting a fleet 1000 automobiles in fifteen different locations." In the latter case, there are two distinct areas of difficulty:

32

1) The statement of the problem is so abstract that it is necessary to break it down into a number of smaller problems, so that each of them may be solved in conjunction with one another.

2) The user may not fully understand all of the implications of the request and may not be able to specify completely the various required activities of the systems and the various tasks that must be accomplished. This situation is particularly likely when the user has little experience with computer systems or with the problem being addressed; there is often a period of time in which the user does not fully understand all of the various possibilities which can be accomplished with a given set of machine-stored data.

These two areas are actually closely related. It is necessary for the software designer to refine the abstract statement of the problem into more and more detailed statements, and to formulate the concepts precisely. In so doing, the designer will learn the precise set of tasks which the user wishes to accomplish in the context of solving the broad problem. At the same time, the user will gain an understanding of what is possible in terms of machine processing of the data and will be able to define specific needs more clearly. The process of refining the problem into a number of smaller tasks is valuable for both the system designer and the user. It is often the case that major changes in requirements will be made at this stage.

For example, a hospital pharmacy may desire to have an information system to keep track of patient medications, drug inventories, and pharmacy administration. In defining the system, the pharmacist/user can observe that the availability of medication information makes it possible to perform some checks concerning drug-drug interactions or medication levels in addition to the other system tasks. While this added requirement makes a significant change in the desired operation of the system, it has been identified at the problem definition phase rather than at some later point where there would be a greater impact on the system design and structure.

It is often the case that problems and requirements are stated ambiguously, making it impossible to develop a program which meets the stated needs. These problems are, to an extent, inherent in natural language and in situations where all of the possibilities have not been fully explored. Proper discipline, however, requires that a clear statement of the problem and/or system requirements be obtained prior to proceeding further.

This is an area which has received relatively little attention in software engineering; in fact, many people do not believe that this process of requirements definition is part of software engineering. However, it is so basic to the software creation process that it must be regarded as the first step of the software life cycle, and the first stage of the design process. One of the most promising efforts to formalize this stage of design is the Problem Statement Language of Teichroew [2], although there are a number of other attempts as well (see [3], for example).

Complete Problem Specification

The process of gaining complete understanding of the problem leads to the development of a complete specification of the task(s) to be performed by the program. Once the concepts are formulated and the problem is clearly delineated, the next objective is to write a functional specification of the program. This specification will state what the program will do, how it will be used by its users, what requirements in terms of performance may exist, and any other information which clarifies the objectives. In some instances, it is also necessary to state what the program should not do; it is occasionally important that the program do only what is specified and no more.

The specification for a program may be short as a single sentence or may fill volumes, as might be the case for a program to perform air traffic control, for example. The specification may be expressed using a mathematical formalism or, more commonly, using less formal prose. If the software developer is working under contract to a customer, it is essential that both parties come to a formal agreement at an early stage as to the nature of the finished product. The user must determine that the specified program meets the needs, and the software developer must use this specification to proceed with the design of the system.

A complete and accurate specification is an important part of proper software development discipline for several reasons. First, it is a checkpoint at which everyone can agree upon the functions to be performed by the program. Second, it serves as a reference point from which a complete design can be developed. The specification provides valuable information about program operation that will strongly influence design of program and data structures. One must take care, though, to make certain that the specification does not impose unnecessary constraints upon the design. Third, without such a specification, it becomes impossible to verify, either formally or informally, that a program does what it is supposed to do. Finally, the specification of a program is important in planning for program testing, so that it may be determined whether the program indeed performs as desired.

Gerhart and Yelowitz [4] have shown that errors in specifications can lead to incorrect programs. Balzer [5] has cited some of the sources of difficult specifications and some of the problems of dealing with them, particularly within the context of automatic programming.

The difficulty of writing a specification varies sharply according to the nature of the task being specified. If one is writing a compiler for a programming language, for example, a formal definition of the language, including syntax and semantics, is a valuable starting point. This formal definition, when supplemented by certain information concerning machine-dependencies for the host computer, provides a good specification. For many other programs, though, particularly for control programs, it is much more difficult to write a complete specification, since it is often difficult to envisage all of the possibilities which may arise during program execution. Hence, it is difficult to specify what is to be done in all cases.

Although the need for good specifications is now becoming widely recognized, there are still very few general techniques for writing them and few automated aids available. Liskov and Zilles have discussed specification techniques as they apply to data objects [6]. Wulf, London, and Shaw have formally incorporated specifications into the definition of a form (an abstract data type) in ALPHARD [7]. This work is directed toward the development of methods for formally specifying the semantics of a program using a precise notation rather than the informal prose which is now generally used. Overall, however, there is still no comprehensive methodology for software specifications. There is need for additional work in this area, since specification is a key step in the software life cycle.

Problem-Solving and Software Design

Dijkstra has observed that many of our problems in programming stem from "unmastered complexity" [8], trying to deal with a problem which is too complex for the programmer to understand completely. One of the key goals of problem-solving, then, is to be able to decompose a large problem into a number of smaller problems which can be solved separately, in such a way that the solutions can be assembled to yield a solution to the original problem. These techniques are familiar in mathematics, for example, where the proof of a theorem is often preceded by the proofs of numerous lemmas. The availability of the results of the lemmas simplifies the proof of the theorem.

In the software development process, the complete specification of the task must be followed by the design of a solution to the problem(s) stated. Once this step has been completed, it then becomes possible to write the program to carry out the solution. In short, proper programming discipline demands that the problem(s) be solved before the program is written. It is very difficult, except for trivial programs, to go directly from the problem specification to executable program code. This phase of problem-solving

is the major part the desi stage.

This sequence of problem-solving, followed by programming, has been the focus of a great deal of research into programming methodology. Although it has unfortunately lost much of its original meaning, the term "structured programming" was intended to refer to systematic program design, followed by coding, with appropriate care at each stage.

The process of program design involves more than problem-solving. In addition to solving the problem at hand, the software signer must consider a diversity of factors luding performance requirements, space limitations, and the eventual coding process. The problem solution must be embedded in a program structure which represents a workable compromise among all of these varied considerations. The design process must be deliberate and carried out in such a manner as to make the reasons for design decisions apparent and to minimize the problems involved in modifying the resulting program. The complete design must be validated against the program specification to make certain that the design represents a solution to the specified task.

There have been a number of design methodologies advanced. The terms "top-down design", "hierarchical decomposition", and "stepwise refinement" [9,10], have been used interchangeably to refer to the process of taking a big problem and breaking it into a number of smaller subordinate problems. At each "level of abstraction", lower-level details are omitted, so that the highest levels present a broad overview of the design with increasing detail at lower levels. These details are invisible at the higher levels.

Another important design technique is "modular decomposition" [11,12,13], the key technique used to isolate subproblems from another. Each module is constructed in such a way that external modules know the functional characteristics of the module, along with the form and content of input and output variables which serve as the interfaces for the module, but know nothing of the actual detailed structure, i.e., code data structures, of the module. Each le may then be handled separately. This modular structure is particularly valuable for program construction, since it makes it possible for different programmers to work independently on different modules of a large program once the modular structure has been established.

Modularity is also valuable for program modifications. Programs can be optimized by replacing modules with other modules which are functionally equivalent, but which execute more efficiently, for example. Changes to the module will not cause any unwanted side effects, so that required modifications are minimized in properly modularized programs.

This top-down approach to problem-solving is valuable, since it serves to provide a structure which facilitates intel[illegible]ual mastery of the [illegible] structure of the problem solution is th[illegible] reflected in the resulting program structure. Hierarchical program structures are natural and well-suited to many programming languages. In short, this approach simplifies the step of transforming the problem solution into a running program.

Unfortunately, the process of design is not as straightforward as the foregoing discussion might lead one to believe. Parnas and Siewiorek have observed [14] that it is often infeasible to derive a design from a specification using this "outside in" approach. Software design is often constrained by lower-level details, including information about the "inside." Design of an operating system, for example, is sharply influenced by the architecture of the host machine. One may specify an operating system in the abstract and then be unable to realize that specification because of the hardware constraints. There are also many situations in which the use of some "bottom up" design is appropriate, such as with the use of existing library routines. In short, successful designs tend to exhibit both "top down" and "bottom up" characteristics.

Although it is difficult to state any firm guidelines for software design techniques, an overall design discipline is an important part of the program creation process. First, the design must be completed to a level of detail that makes it possible to evaluate the quality of the design and to identify any remaining design problems. The completed design should show the various modules, their interfaces, and the operations to be performed within each module, expressed in an unambiguous way. The use of a "pseudo-code" such as PDL [15] is quite valuable for this task. Next, the design must be compared with the specification, to make certain that the program design meets the original goals for the program.

It is often the case that the program must be redesigned at this stage in order to bring the design into agreement with the specification. Finally, this detailed design sometimes points out potential areas of implementation difficulty. It may even be necessary to modify the specification if it is determined that the design cannot be feasibly implemented. Freeman has suggested [16] some techniques which can aid in this process of design review.

It should be noted that no coding should be done to this point. Since it is often necessary to redo the specification and design, any code written previously might very well be useless. One of the most difficult aspects of programming discipline for the experienced programmer is to refrain from writing code at the early stages of software development. Most programmers have acquired bad habits when they first learned to program, often because they were not required to develop the necessary discipline in writing these programs. They were usually able to write them directly from the problem statement; they retain this tendency to write code immediately even as they progress to more complex problems.

The completed design now serves as the basis for the program. The programming stage involves the transformation of the problem solution, i.e., the design, into code which executes reliably and efficiently. A well-structured design should lead to a well-structured program.

The concept of what constitutes a "good" program has changed rather sharply over the past few years. Among the earlier ideals of programs was compactness, both of source code and of object code. Programmers routinely exploited a variety of tricks to achieve this objective, often producing cryptic and convoluted programs. This notion of compactness of source code as an ideal has now been partially discredited in favor of program readability.

Similarly, subtle programming tricks were used in the name of efficiency in order to save a few milliseconds or a few words of primary memory. While efficiency is still an important programming consideration, particularly for those programs which will be used heavily, it must be balanced against other ideals of program correctness and comprehensibility. The efficiency of a program is irrelevant if it does not yield the correct answers.

This concern about program correctness and the techniques for verifying that programs (as well as designs) met their specifications was a key factor in the recognition of the need for good programming discipline. For example, it was observed that programs are more readable and easier to verify if the dynamic flow of program control closely resembles the static appearance, i.e., the listing, of the program [17]. This observation led to lengthy debates over the use of the GO TO statement in programming [18,19]. It also led to a great deal of research into programming languages and their design, including modifications to existing languages, design of new languages and language features, and the advancement of general criteria for programming language design [20,21,22,23].

Much has been written about programming style [24,25], those techniques which contribute to the readability and understandability of programs. Among the most common recommendations are good internal program documentation, use of mnemonic variable names, minimization of the use of GO TO statements, and modular program structure. Of all of the aspects of programming discipline, the concepts of structured coding are the most widely accepted. Many organizations have established programming standards and uniform program structures to enforce coding discipline.

Coding, the construction phase of software development, is now seen to occupy a relatively small portion of the entire process of software design and development. Furthermore, it can be seen that coding is relatively straightforward, given a good design and a good set of programming tools including a programming language which is well-suited for its intended task. Where a great deal of innovation and creativity is possible at the design stage, the coding stage is much more tightly constrained.

Software Quality Assurance -- Testing and Verification

Following program construction, it is necessary to determine whether the program works correctly and reliably. It is important to note here a distinction between correctness and reliability. Correctness is a programmer's notion that the program performs according to the specification. Reliability is more commonly a user's notion that the program does what is desired when it is desired. Ideally, these notions are equivalent. However, correct programs may be unreliable if they run on unreliable computer systems or if they are particularly unforgiving of user errors.

There are two major approaches which can be taken toward determining whether or not a program works properly: testing and verification. Testing involves the generation of an adequate number of data sets so as to exercise as many portions of the program as possible in order to determine whether or not the program performs properly. In some instances, testing is used to determine the operation of the system under load. Such an approach is required, for example, in measuring response times and in studying a multiprogramming operating system.

Verification, by contrast, involves the construction of a rigorous proof which confirms that the program meets its specification. In general, verification is a formal approach to determining correctness, while testing is an informal approach.

Both testing and verification rely largely on *ad hoc* methods. Nonetheless, they play a major role in programming discipline. Programs should be constructed with the notion of a proof in mind. Individual program modules can be verified within a large system; confidence in the correctness of a large program can be gained through confidence in the correctness of the modules of which it is composed. Good has termed this approach "provable programming" [26].

The testing methodology for a program should be developed beginning with the specification stage. The various cases that must be handled become apparent at a very early stage of the software development process. In many settings, a program specification is given to an outside party who develops a set of test data for the program. The eventual acceptance of the program by the user is determined by its ability to perform properly on the test data.

Top-down testing involves use of a testing methodology which parallels the design and programming approach. In this way, testing proceeds as the program is developed, making it possible to test the

jor interfaces first and to distribute most of the testing throughout the programming phase, with a resultant decrease in total time for program construction. Yourdon [27] outlines this philosophy in greater detail.

Testing should not be confused with bugging. Debugging is an activity which is used in an attempt to locate known software errors, while testing is an orderly procedure carried out in an attempt to determine if such errors exist. In general, debugging ld be completed before testing is begun. Conceptually, bugs arise from two major sou : errors in the process of transforming the problem solution into a program, normally caused by improper use of programming language, and errors in the problem solution itself. Only the first of these two sources is partially amenable to traditional debugging techniques; the pre ice of bugs should be a caution to the programmer to review the entire programming process, rather than proceeding in an undirected search for the bug(s).

In the past, many informal rules have been applied for the selection of test data, ge ally involving the boundary cases and the most common paths of control flow. Testing was used to find program errors, rat than to show that none existed. Recently, however, an attempt is being made to demonstrate program correctness through testing. Gerhart and Goodenough have developed the rudiments of a theory of test data selection [28]. Systems have also been c ucted which execute programs symbolically, in order to systematically test all paths of control flow [29,30].

Although a few automated aids for verification have been developed [31,32], they are generally limited to relatively l programs and cannot easily be applied to large systems. One of the problems in this regard is that a formal verification requires a formal specification in order to be meaningful. Given the absence of such formal specifications for a program, one cannot hope to verify it. One of the key rea for the presence of formal s. ifications in ALPHARD [7] is to provide a basis for program verification.

Software Quality Assurance -- Performance

The assurance of program quality involves not only correctness, but also efficiency. Although correctness concerns are paramount in software development, program performance cannot be neglected. Many programs, particularly those involving elements of real-time control, have severe performance constraints. Similarly, programs which may be used for lengthy or repetitious computation must perform efficiently. While a bubble sort may indeed correctly sort a sequence of numbers, many more effective sorting techniques exist; one such technique should almost certainly be used instead of a bubble sort. Elementary analysis of algorithms should be used during the program design stage to evaluate alternative approaches to construction of an algorithm.

At the design and coding stage, one must be careful to select an effective gene approach to the problem solution, to allocate space carefully, and to avoid unnecessary duplication of computation. However, more subtle program modifications should be made once the program is running correctly, with the possible exception of certain critical routines which have specific execution time limitations.

Refinement of performance is now generally treated as secondary to reliability considerations. It is well recognized that most computer programs spend a large percentage of their execution time in a small percentage of the code. Thus, the ideal approach to the problem of program efficiency is to construct first a program which performs correctly, then to locate those program segments which are most heavily used, and finally to optimize the execution of those program segments to improve overall performance. If the guidelines concerning program modularity have been followed, it becomes possible to rewrite modules to improve performance and to test the program using the different modules to compare performance.

Performance analysis and optimization is often much more subtle than it might appear. Analysis of algorithms and program reorganization are useful techniques, but they do not fully take the program execution environment into consideration. Many programs will be used in a multiprogramming environment or a virtual memory environment (or both). In those cases, the performance of a program, with respect to response time, for example, may be strongly affected by the current system load, which typically varies sharply during a day. Programs which involve conversational access to a data base are especially subject to such variations, making it particularly difficult to predict or optimize performance.

Quality assurance is the final step of the development process. Because the code has already been written, there is often the tendency to place less emphasis than necessary on this step. However, it is essential to follow the programming discipline through to this point. Although the software designer and programmer can have a certain amount of faith in the program, it is only through testing, verification, and optimization that this faith can be justified. This stage of quality assurance should be treated as part of the construction stage in the same way as many manufacturing concerns routinely perform quality control on their products before shipping them to distributors and customers.

In many ways, the emphasis on software quality has been responsible for all of the stages already mentioned. It has motivated much of the research into programming methodology and software engineering and has resulted in the development of the discipline described here. It should continue to be a dominating influence in the development of improved techniques for testing and verification of programs.

One of the predominant underlying themes of discipline in software design and elopment is the need to commit all major ps and decisions to writing. In any software development effort, a large amount of effort is expended in communication among people. Verbal communication is not only subject to misinterpretation, but is easily forgotten. As a result, it is important to maintain good documentation throughout a project. The specification must be used many times throughout the software development effort: to perform and validate the design, to write the input and output routines in the program to conform to reporting formats, to develop testing and verification procedures, and to evaluate the need to change the specifications in response to new considerations, just to name a few of the major uses. These tasks could not be carried out in the absence of a good written specification.

Likewise, the design should be clearly written. A good design document is not only essential for writing the program, but also for maintenance of the program over time. The design document provides an insight into the overall program structure and logic, as well as indicating some of the alternatives that may have been considered and rejected.

The need for documentation extends into the actual program code. Because programs will be used for a period of time and will go through numerous changes during their lifetimes, they will be read by more than one person. The original programmers must provide as much information as possible to assist others (and possibly themselves) in comprehending the program logic at a later date.

User documentation is crucial as well. The poor quality of user documents in the computing field is almost legendary. They tend to be heavily laden with acronyms and jargon, along with references to other inscrutable documents. In many cases, the best approach is to develop user documentation from the program specification. Preliminary user documents can be available before program design is completed, so that potential users have an opportunity to get a feeling for the operation of the program before it is built and to recommend changes in the specification and/or design.

Documentation is particularly important when such a change is suggested or made. The reasons for the desired change should be documented. If the change is to be made, it is necessary to revise the specification to reflect this change and then to alter the design as needed.

This aspect of documentation points out the need for some form of document control mechanism. When a change is made in a document, the revision should be made available to everyone who has access to the original document. Otherwise, different individuals may be working with inconsistent documents, [illegible] disaster.

This need for good documentation practices is receiving considerable attention. Among the many aids to documentation are HIPO (Hierarchy-plus-Input-Process-Output) charts [33] and the methodology of Chief Programmer Teams [34], in which a single individual has responsibility for document control, including code. The ability to communicate clearly is recognized as a key quality for software engineers [35,36].

The Discipline of Discipline

Discipline in software design and development has many facets. However, they are well summed up by the need to follow a set of rules and procedures throughout the creation of a program, to review earlier decisions in the light of new information, and to record all of the decisions thoroughly to achieve effective communication.

Beyond that, however, it is important to note that the software creation process is nonlinear. Various steps occur in parallel and there are occasions where the procedure must back up before it can move forward. The phases of problem definition, program specification, design, and coding must each be completed before proceeding to the subsequent phase, although it may be necessary to return when a specification is found to be incomplete or incorrect or when a design cannot be effectively implemented. However, the tasks of documentation, testing, verification, and performance review occur in parallel with these other phases. All of these steps must be carried out as part of the software development discipline.

One remaining aspect of the discipline is to maintain the discipline, even in the face of adversity. If this methodology is used and the resulting program is inefficient, too large, or behind schedule, there is a temptation to abandon these techniques or to try cutting corners. While software engineering practices are not a panacea for programming problems, the return to undisciplined practices can be catastrophic and will almost certainly lead to an inferior final product. The shortcomings of any program constructed using these techniques is most commonly due to unrealistic predictions or expectations or to improper application of the methodology, rather than to the methodology itself.

CONCLUSION -- TOWARD IMPROVED DISCIPLINE

Improvement in programming discipline requires both increased usage of the methodologies associated with systematic program construction and the development of new techniques based on this experience. Many of the ideas advanced in software engineering have not been widely used or thoroughly evaluated. There is a need to quantify some of the subjective feelings. What, for example, is the best way to measure software reliability? Can anything meaningful be said about changes in

programmer productivity using this recommended discipline? Engineering is generally defined as the application of pure sci ; if we are to achieve the reality of engi ed software, then we must seek out and identify some basic principles of ftware science."

Not only is it important to develop and ex these concepts, but it is also important to be able to teach them [8]. The quality of software production will only improve when these programming methodologies are more widely used. Thus, some form of technology transfer is required in order to increase awareness of these ideas and to get them accepted and routinely used. From an industrial standpoint, the sources of this technology transfer can be books, survey articles, consultants, short courses, or new employees who have been exposed to these ideas in school.

From an educational standpoint, it is es ial that good programming habits be taught to students from the outset of their exposure to programming so that they never acquire undesirable and undisciplined practices. Even introductory programming courses should give emphasis to the notions of problem-solving, design, and programming styl rather than concentrating on the mastery of a programming language [37]. For those students who intend to work on software elopment in industry or government, software engineering education should give the student some exposure to the problems associated with the development of large software systems in such settings [38].

These changes will not occur overnight. spite the many rapid changes in computing, software development practices have changed slowly. Furthermore, there is a mammoth in tment in existing software to which techniques cannot be easily applied. If a programmer's job involves maintenance of a large assembly language program written in a disorganized way (and probably altered n ous times), few of the ideas advanced e will be of practical use in carrying out that job, until the program is thrown out and rewritten. In short, it is nearly impossible to impose discipline post hoc; the discipline must be inherent in program development in order to be useful.

Thus, improved discipline will only come with time. The required educational process will take many years, since it is necessary to retrain the teachers before the students be properly taught. Meanwhile, new programming techniques will be evaluated so that they can be refined into effective methodologies, and better tools for software development will be produced. Eventually, then, an increased percentage of programs will exhibit the results of a deliberate, ll-organized, and coherent approach to their design and development. At some later point, perhaps the design and construction of programs can be understood so thoroughly that one can manage their construction in the same way that one manages the construction of physical structures.

REFERENCES

[1] Naur, P. and B. Randell (eds.) Software Engineering. Brussels: NATO Science Committee, 1969.

[2] Teichroew, D., "PSL/PSA -- A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," Proc. 2nd Int'l Conf. on Software Engineering, 1976.

[3] Bell, T. and D. Bixler, "A Flow-Oriented Requirements Statement Language," Proc. 1976 MRI Symposium. Brooklyn: Polytechnical Institute of New York, 1976.

[4] Gerhart, S. and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, vol. SE-2, no. 3 (September, 1976).

[5] Balzer, R.M., "Imprecise Program Specification," USC Information Sciences Institute Research Report RR-75-36, December, 1975.

[6] Liskov, B. and S. Zilles, "Specification Techniques for Abstract Data Types," IEEE Transactions on Software Engineering, vol. SE-1, no. 1 (March, 1975), pp. 7-19.

[7] Wulf, W., R. London, and M. Shaw, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology," Carnegie-Mellon University Computer Science Department Technical Report, 1976.

[8] Dijkstra, E., "Programming Methodologies: their Objectives and their Nature," in Structured Programming, ed. D. Bates. Maidenhead, England: Infotech International, 1976, pp. 203-216.

[9] Wirth, N., "Program Development by Stepwise Refinement," Comm. ACM, vol. 14, no. 4 (April, 1971), pp. 221-227.

[10] Mills, H.D., "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, ed. R. Rustin. Englewood Cliffs, N.J.: Prentice-Hall, 1971, pp. 41-55.

[11] Parnas, D., "A Technique for Software Module Specification with Examples," Comm. ACM, vol. 15, no. 5 (May, 1972), pp. 330-336.

[12] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM, vol. 15, no. 12 (December, 1972), pp. 1053-1058.

[13] Liskov, B., "A Design Methodology for Reliable Software Systems," Proc. AFIPS 1972 FJCC, vol. 41, part 1, pp. 191-200.

[14] Parnas, D. and D. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," Comm. ACM, vol. 18, no. 7 (July, 1975), pp. 401-408.

[15] Caine, S. and E. Gordon, "PDL -- A Tool for Software Design," Proc. AFIPS 1975 NCC, vol. 44, pp. 271-276.

[16] Freeman, P., "Toward Improved Review of Software Design," Proc. AFIPS 1975 NCC, vol. 44, pp. 329-334.

[17] Dijkstra, E., "GO TO Statement Considered Harmful," Comm. ACM, vol. 11, no. 3 (March, 1968), pp. 147-148.

[18] Wulf, W., "A Case Against the GO TO," Proc. 1972 ACM Conference, pp. 791-797.

[19] Knuth, D., "Structured Programming with GO TO Statements," ACM Computing Surveys, vol. 6, no. 4 (December, 1974), pp. 261-301.

[20] Wasserman, A. (ed.), "Special Issue on Programming Language Design," ACM SIGPLAN Notices, vol. 10, no. 7 (July, 1975).

[21] Hoare, C.A.R., "Hints on Programming Language Design," Stanford University Computer Science Department Technical Report CS-73-403, December, 1973.

[22] Wirth, N., "The Programming Language PASCAL," Acta Informatica, vol. 1, no. 1 (1971), pp. 35-63.

[23] Cheatham, T. E., Jr., "The Recent Evolution of Programming Languages," Proc. IFIP Congress 71. Amsterdam: North-Holland, 1972, pp. 298-313.

[24] Kernighan, B. and Plauger, P., The Elements of Programming Style. New York: McGraw-Hill, 1974.

[25] McCracken, D. A Simplified Guide to Structured COBOL Programming. New York: John Wiley and Sons, 1976.

[26] Good, D., "Provable 'Programming," Proc. 1975 Int'l Conf. on Reliable Software, pp. 411-419.

[27] Yourdon, E. Techniques of Program Structure and Design. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

[28] Goodenough, J. and S. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, vol. SE-1, no. 2 (June, 1975), pp. 156-173.

[29] King, J., "A New Approach to Program Testing," Proc. 1975 Int'l Conf. on Reliable Software, pp. 228-233.

[30] Boyer, R.S., B. Elspas, and K. Levitt, "SELECT -- A Formal System for Testing and Debugging," Proc. 1975 Int'l Conf. on Reliable Software, pp. 234-245.

[31] Good, D., R. London, and W. Bledsoe, "An Interactive Program Verification System," IEEE Transactions on Software Engineering, vol. SE-1, no. 1, pp. 59-67.

[32] Suzuki, N., "Verifying Programs by Algebraic and Logical Reduction," Proc. 1975 Int'l Conf. on Reliable Software, pp. 473-481.

[33] Stay, J.F., "HIPO and Integrated Program Design," IBM Systems Journal, vol. 15, no. 2 (1976), pp. 143-154.

[34] Baker, F.T., "Structured Programming in a Production Programming Environment," Proc. 1975 Int'l Conf. on Reliable Software, pp. 172-183.

[35] Boehm, B., "Software Engineering Education: Some Industry Needs," in Software Engineering Education: Needs and Objectives, ed. A. Wasserman and P. Freeman. Berlin: Springer-Verlag, 1976, pp. 13-18.

[36] Freeman, P., A. Wasserman, and R. Fairley, "Essential Elements of Software Engineering Education," Proc. 2nd Int'l Conf. on Software Engineering.

[37] Wasserman, A. and P. Freeman, "Software Engineering Concepts and Computer Science Curricula," submitted for publication.

[38] Wasserman, A. and P. Freeman (eds.) Software Engineering Education: Needs and Objectives. Berlin: Springer-Verlag, 1976.

A design methodology for reliable software systems*

by B. H. LISKOV**

The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

Any user of a computer system is aware that current systems are unreliable because of errors in their software components. While system designers and implementers recognize the need for reliable software, they have been unable to produce it. For example, operating systems such as OS/360 are released to the public with hundreds of errors still in them.[1]

A project is underway at the MITRE Corporation which is concerned with learning how to build reliable software systems. Because systems of any size can always be expected to be subject to changes in require- the project goal is to produce not only reliable software, but readable software which is relatively easy to modify and maintain. This paper describes a design methodology developed as part of that project.

Rationale

Before going on to describe the methodology, a few words are in order about why a design methodology approach to software reliability has been selected.† The unfortunate fact is that the standard approach to building systems, involving extensive debugging, has not pr successful in producing reliable software, and there is no reason to suppose it ever will. Although improvements in debugging techniques may lead to the detection of more errors, this does not imply that all errors will be found. There certainly is no guarantee of this implicit in debugging: as Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence."[2]

In order for testing to guarantee reliability, it is necessary to insure that all relevant test cases have been checked. This requires solving two problems:

(1) A complete (but minimal) set of relevant test cases must be identified.
(2) It must be possible to test all relevant test cases; this implies that the set of relevant test cases is small and that it is possible to generate every case.

The solutions to these problems do not lie in the domain of debugging, which has no control over the sources of the problems. Instead, since it is the system design which determines how many test cases there are and how easily they can be identified, the problems can be solved most effectively during the design process: The need for exhaustive testing must influence the design.

We believe that such a design methodology can be developed by borrowing from the work being done on proof of correctness of programs. While it is too difficult at present to give formal proofs of the correctness of large programs, it is possible to structure programs so that they are more amenable to proof techniques. The objective of the methodology presented in this paper is to produce such a program structure, which will lend itself to *informal* proofs of correctness. The proofs, in addition to building confidence in the correctness of the program, will help to identify the relevant test cases, which can then be exhaustively tested. When exhaustive testing is combined with informal proofs, it is reasonable to expect reliable software after testing is complete. This expectation is borne out by at least one experiment performed in the past.[3]

The scope of the paper

A key word in the discussion of software reliability is "complex"; it is only when dealing with complex sys-

* This work was supported by Air Force Contract No. F19(628)-71-C-0002.
** Present Address—Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.
† The material in this section is covered in much greater detail in Liskov and Towster.[3]

...hat reliability becomes an acute problem. A two-fo... definition is offered for "complex." First, there are many system states in such a system, and it is difficult to organize the program logic to handle all states correctly. Second, the efforts of many individuals must be coordinated in order to build the system. A design methodology is concerned with providing techniques which enable designers to cope with the inherent logical complexity effectively. Coordination of the efforts of individuals is accomplished through management techniques.

The fact that this paper only discusses a design methodology should not be interpreted to imply that management techniques are unimportant. Both design methodology and management techniques are essential to the successful construction of reliable systems. It is customary to divide the construction of a software system into three stages: design, implementation, and testing. Design involves both making decisions about what precisely a system will do and then planning an overall structure for the software which enables it to perform its tasks. A "good" design is an essential first step toward a reliable system, but there is still a long way to go before the system actually exists. Only management techniques can insure that the system implementation ...to the structure established by the design and that ...austive testing is carried out. The management techniques should not only have the form of requirements placed on personnel; the organization of personnel is also important. It is generally accepted that the organizational structure imposes a structure on the system being built.[4] Since we wish to have a system structure based on the design methodology, the organizational structure must be set up accordingly.*

CRITERIA FOR A GOOD DESIGN

The design methodology is presented in two parts. This section defines the criteria which a system design should satisfy. The next section presents guidelines intended to help a designer develop a design satisfying the criteria.

To reiterate, a complex system is one in which there are so many system states that it is difficult to understand how to organize the program logic so that all states will be handled correctly. The obvious technique to apply when confronting this type of situation is "divide and rule." This is an old idea in programming and is known as modularization. Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas:[3] "The connections between modules are the assumptions which the modules make about each other." Modules have connections in control via their entry and exit points; connections in data, explicitly via their arguments and values, and implicitly through data referenced by more than one module; and connections in the services which the modules provide for one another.

Traditionally, modularity was chosen as a technique for system production because it makes a large system more manageable. It permits efficient use of personnel, since programmers can implement and test different modules in parallel. Also, it permits a single function to be performed by a single module and implemented and tested just once, thus eliminating some duplication of effort and also standardizing the way such functions are performed.

The basic idea of modularity seems very good, but unfortunately it does not always work well in practice. The trouble is that the division of a system into modules may introduce additional complexity. The complexity comes from two sources: functional complexity and complexity in the connections between the modules. Examples of such complexity are:

(1) A module is made to do too many (related but different) functions, until its logic is completely obscured by the tests to distinguish among the different functions (functional complexity).
(2) A common function is not identified early enough, with the result that it is distributed among many different modules, thus obscuring the logic of each affected module (functional complexity).
(3) Modules interact on common data in unexpected ways (complexity in connections).

The point is that if modularity is viewed only as an aid to management, then any ad hoc modularization of the system is acceptable. However, the success of modularity depends directly on how well modules are chosen. We will accept modularization as the way of organizing the programming of complex software systems. A major part of this paper will be concerned with the question of how good modularity can be achieved, that is, how modules can be chosen so as to minimize the connections between them. First, however, it is necessary to give a definition of "good" modularity. To emphasize the requirement that modules be as disjoint as possible, and because the term "module" has been used so often and so diversely, we will discard it and define modularity as the division of the system into

* Management techniques intended to support the design methodology proposed in this paper are described by Liskov.[5]

"partitions." The definition of good modularity will be on a synthesis of two techniques, each of which addresses a different aspect of the problem of constructing reliable software. The first, levels of abstraction, permits the development of a system design which copes with the inherent complexity of the system effectively. The second, structured programming, insures a clear understandable representation of the design in the system software.

Levels of abstraction

Levels of abstraction were first defined by Dijkstra.[5] They provide a conceptual framework for achieving a and logical design for a system. The entire system is conceived as a hierarchy of levels, the lowest levels being those closest to the machine. Each level supports an important abstraction; for example, one level might support segments (named virtual memories), while another (higher) level could support files which consist of several segments connected together. An example of a file system design based entirely on a hierarchy of levels can be found in Madnick and Alsop.[6]

Each level of abstraction is composed of a group of related functions. One or more of these functions may be referenced (called) by functions belonging to other levels; these are the external functions. There may also internal functions which are used only within the to perform certain tasks common to all work being performed by the level and which cannot be referenced other levels of abstraction.

Levels of abstraction, which will constitute the parti- of the system, are accompanied by rules governing of the connections between them. There are two important rules governing levels of abstraction. The first concerns resources (I/O devices, data): each level resources which it owns exclusively and which other levels are not permitted to access. The second involves the hierarchy: lower levels are not aware of the existence of higher levels and therefore may not refer to them in way. Higher levels may appeal to the (external) f ions of lower levels to perform tasks; they may also appeal to them to obtain information contained in the resources of the lower levels.*

* In the Madnick and Alsop paper referenced earlier, the hierarchy of levels is strictly enforced in the sense that if the third level wishes to make use of the services of the first level, it must do so h the second level. This paper does not impose such a strict requirement; a high level may make use of a level several steps below it in the hierarchy without necessarily requiring the assistance of intermediate levels. The "THE" system[5] and the Venus system[16] contain examples of levels used in this way.

Structured programming

Structured programming is a programming discipline which was introduced with reliability in mind."[12] Although of fairly recent origin, the term "structured programming" does not have a standard definition. We will use the following definition in this paper.

Structured programming is defined by two rules. The first rule states that structured programs are developed from the top down, in levels.* The highest level describes the flow of control among major functional *components* (major subsystems) of the system; *component names* are introduced to represent the components. The names are subsequently associated with code which describes the flow of control among still lower-level components, which are again represented by their component names. The process stops when no undefined names remain.

The second rule defines which control structures may be used in structured programs. Only the following control structures are permitted: concatenation, selection of the next statement based on the testing of a condition, and iteration. Connection of two statements by a *goto* is not permitted. The statements themselves may make use of the component names of lower-level components.

Structured programming and proofs of correctness

The goal of structured programming is to produce program structures which are amenable to proofs of correctness. The proof of a structured program is broken down into proofs of the correctness of each of the components. Before a component is coded, a specification exists explaining its input and output and the function which it is supposed to perform. (The specification is defined at the time the component name is introduced; it may even be part of the name.) When the component is coded, it is expressed in terms of specifications of lower level components. The theorem to be proved is that the code of the component matches its specifications; this proof will be given based on axioms stating that lower level components match their specifications.

The proof depends on the rule about control structures in two important ways. First, limiting a component to combinations of the three permissible control structures insures that control always returns from a component to the statement following the use of the

* The levels in a structured program are not (usually) levels of abstraction, because they do not obey the rule about ownership of resources.

component name (this would not be true if *goto* statements were permitted). This means that reasoning about the flow of control in the system may be limited to the flow of control as defined locally in the component being proved. Second, each permissible control structure is associated with a well-known rule of inference: concatenation with linear reasoning, iteration with induction, and conditional selection with case analysis. These rules of inference are the tools used to perform the proof (or understand the component).

Structured programming and system design

Structured programming is obviously applicable to system implementation. We do not believe that by itself it constitutes a sufficient basis for system design; rather we believe that system design should be based on identification of levels of abstraction.* Levels of abstraction provide the framework around which and within which structured programming can take place. Structured programming is compatible with levels of abstraction because it provides a comfortable environment in which to deal with abstractions. Each structured program component is written in terms of the names of lower-level components; these names, in effect, constitute a vocabulary of abstractions.

In addition, structured programs can replace flowcharts as a way of specifying what a program is supposed to do. Figure 1 shows a structured program for the top level of the parser in a bottom-up compiler for an

```
begin
integer relation;
boolean must_scan;
string symbol;
stack parse_stack;
must_scan := true;
push(parse_stack, eol_entry);
while not finished(parse_stack) do
  begin
  if must_scan then symbol := scan_next_symbol;
  relation := precedence_relation(top(parse_stack), symbol);
  perform_operation_based_on_relation(relation, parse_stack,
                                      symbol, must_scan)
  end
end
```

Figure 1—A structured program for an operator precedence parser

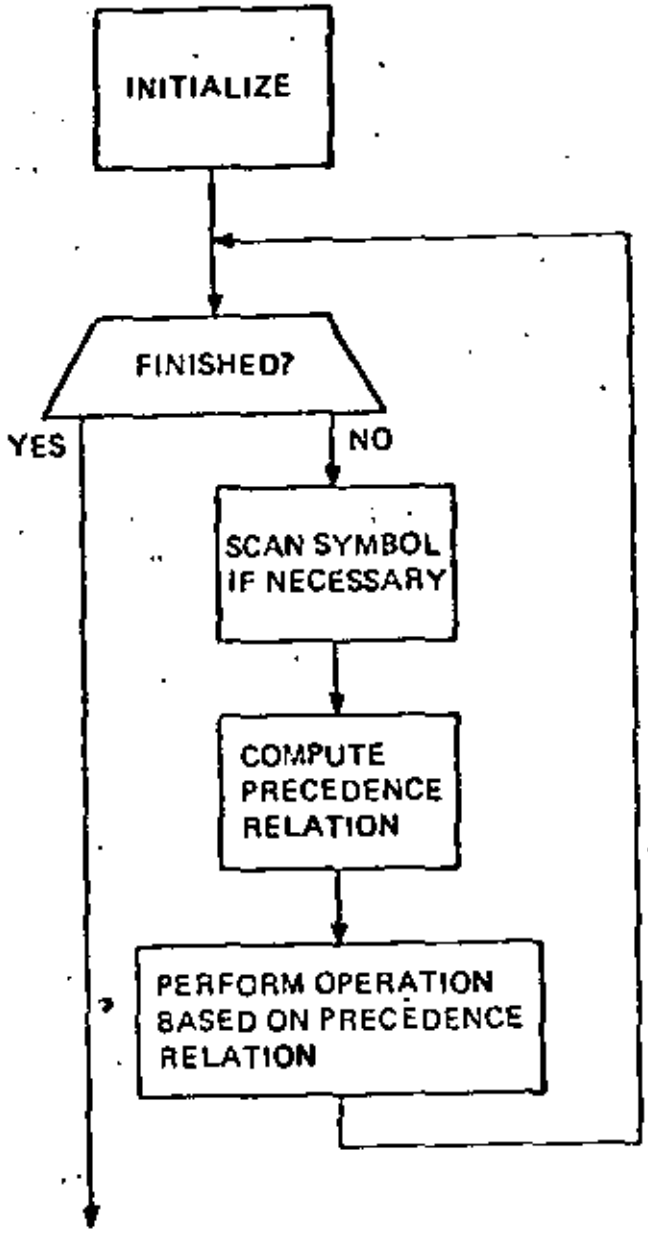


Figure 2—Flowchart of an operator precedence parser

operator precedence grammar, and Figure 2 is a flowchart containing approximately the same amount of detail. While it is slightly more difficult to write the structured program, there are compensating advantages. The structured program is part of the final program; no translation is necessary (with the attendant possibility of introduction of errors). In addition, a structured program is more rigorous than a flowchart. For one thing, it is written in a programming language and therefore the semantics are well defined. For another, a flowchart only describes the *flow of control* among parts of a system, but a structured program at a minimum must also define the data controlling its flow,

* A recent paper by Henderson and Snowden[13] describes an experiment in which structured programming was the only technique used to build a program. The program had an error in it which was the direct result of not identifying a level of abstraction.

so the description it provides is more concrete. In addition, it defines the arguments and values of a referenced component, and if a change in level of abstraction occurs that point, then the data connection between the two components is completely defined by the structured program. This should help to avoid interface errors usually uncovered during system integration.

; definition

We now present a definition of good modularity supporting the goal of software reliability. The system is divided into a hierarchy of partitions, where each partirepresents one level of abstraction, and consists of or more functions which share common resources. At the same time, the entire system is expressed by a structured program which defines the way control among the partitions. The connections between the partitions are limited as follows:

(1) The connections in control are limited by the rules about the hierarchy of levels of abstraction and also follow the rules for structured programs.
(2) The connections in data between partitions are limited to the explicit arguments passed from the functions of one partition to the (external) functions of another partition. Implicit interaction on common data may only occur among functions within a partition.
(3) The combined activity of the functions in a partition support its abstraction and nothing more. This makes the partitions logically independent of one another. For example, a partition supporting the abstraction of files composed of many virtual memories should not contain any code supporting the existence of virtual memories.

A system design satisfying the above requirements is compatible with the goal of software reliability. Since the system structure is expressed as a structured program, it should be possible to prove that it satisfies the system specifications, assuming that the structured programs which will eventually support the functions of the levels of abstraction satisfy their specifications. In addition, it is reasonable to expect that exhaustive testing of all relevant test cases will be possible. Exhaustive testing of the whole system means that each partition must be exhaustively tested, and all combinations of partitions must be exhaustively tested. Exhaustive testing of a single partition involves both testing based on input parameters to the functions in the partition and testing based on intermediate values of state variables of the partition. When this testing is complete, it is no longer necessary to worry about the state variables because of requirement 2. Thus, the testing of combinations of partitions is limited to testing the input and output parameters of the external functions in the partitions. In addition, requirement 3 says that partitions are logically independent of one another; this means that it is not necessary when combining partitions to test *combinations* of the relevant test cases for each partition. Thus, the number of relevant test cases for two partitions equals the sum of the relevant test cases for each partition, not the product.

GUIDELINES FOR SYSTEM DESIGN

Now that we have a definition of good modularization, the next question is how a system modularization satisfying this definition can be achieved. The traditional technique for modularization is to analyze the execution-time flow of the system and organize the system structure around each major sequential task. This technique leads to a structure which has very simple connections in control, but the connections in data tend to be complex (for examples see Parnas[4] and Cohen[3]). The structure therefore violates requirement 2; it is likely to violate requirement 3 also since there is no reason (in general) to assume any correspondence between the sequential ordering of events and the independence of the events.

If the execution flow technique is discarded, however, we are left with almost nothing concrete to help us make decisions about how to organize the system structure. The guidelines presented here are intended to help rectify this situation. First are some guidelines about how to select abstractions; these guidelines tend to overlap, and when designing a system, the choice of a particular abstraction will probably be based on several of the guidelines. Next the question of how to proceed with the design is addressed. Finally, an example of the selection of a particular abstraction within the Venus system[10] is presented to illustrate the application of several of the principles; an understanding of Venus is not necessary for understanding the example.

Guidelines for selecting abstractions

Partitions are always introduced to support an abstraction or concept which the designer finds helpful in thinking about the system. Abstraction is a very valuable aid to ordering complexity. Abstractions are introduced in order to make what the system is doing clearer and more understandable; an abstraction is a conceptual simplification because it expresses what is being done

without specifying how it is done. The purpose of this [illegible]on is to discuss the types of abstractions which [illegible] be expected to be useful in designing a system.

Abstractions of resources

Every hardware resource available on the system will be represented by an abstraction having useful characteristics for the user or the system itself. The abstraction will be supported by a partition whose functions map the characteristics of the abstract resource into the characteristics of the real underlying resource or resources. This mapping may itself make use of several lower partitions, each supporting an abstraction useful in defining the functions of the original partition. It is likely that a strict hierarchy will be imposed on the group of partitions; that is, other parts of the system may only reference the functions in the original partition. In this case, we will refer to the lower partitions as "sub-partitions."

Two examples of abstract resources are given. In an interactive system, "abstract teletypes" with end-of-message and erasing conventions are to be expected. In a multiprogramming system, the abstraction of processes frees the rest of the system from concern about true number of processors.

Abstract characteristics of data

In most systems the users are interested in the structure of data rather than (or in addition to) storage of data. The system can satisfy this interest by the inclusion of an abstraction supporting the chosen data structure; functions of the partition for that abstraction will map the structure into the way data is actually represented by the machine (again this may be accomplished by several sub-partitions). For example, in a file management system such an abstraction might be an indexed sequential access method. The system itself also benefits from abstract representation of data; for example, the scanner in a compiler permits the rest of the compiler to deal with symbols rather than with characters.

Simplification via limiting information

According to the third requirement for good modularization, the functions comprising a partition support only one abstraction and nothing more. Sometimes it is difficult to see that this restriction is being violated, or recognize that the possibility for identification of another abstraction exists.

One technique for simplification is to limit the amount of information which the functions in the partition need to know (or even have access to). An example of such information is the complicated format in which data is stored for use by the functions in the partition (the data would be a resource of the partition). The functions require the information embedded in the data but need not know how it is derived from the data. This knowledge can be successfully hidden within a lower partition (possibly a sub-partition) whose functions will provide requested information when called; note that the data in question become a resource of the lower partition.

Simplification via generalization

Another technique for simplification is to recognize that a slight generalization of a function (or group of functions) will cause the functions to become generally useful. Then a separate partition can be created to contain the generalized function or functions. Separating such groups is a common technique in system implementation and is also useful for error avoidance, minimization of work, and standardization. The existence of such a group simplifies other partitions, which need only appeal to the functions of the lower partition rather than perform the tasks themselves. An example of a generalization is a function which will move a specified number of characters from one location to another, where both locations are also specified; this function is a generalization of a function in which one or more of the input parameters is assumed.

Sometimes an already existing partition contains functions supporting tasks very similar to some work which must be performed. When this is true, a new partition containing new versions of those functions may be created, provided that the new functions are not much more complex than the old ones.

System maintenance and modification

Producing a system which is easily modified and maintained is one of our primary goals. This goal can be aided by separating into independent partitions functions which are performing a task whose definition is likely to change in the future. For example, if a partition supports paging of data between core and some backup storage, it may be wise to isolate as an independent partition those functions which actually know what the backup storage device is (and the device becomes a resource of the new partition). Then if a new device is added to the system (or a current device is removed), only the functions in the lower partition will be affected; the higher partition will have been isolated

such changes by the requirement about data connections between partitions.

to proceed with the design

Two phases of design are distinguished. The very first phase of the design (phase 1) will be concerned with defining precise system specifications and analyzing them with respect to the environment (hardware or software) which the system will eventually exist. The result of this phase will be a number of abstractions which repre- the eventual system behavior in a very general These abstractions imply the existence of parti- but very little is known about the connections between the partitions, the flow of control among the partitions (although a general idea of the hierarchy of partitions will exist), or how the functions of the parti- will be coded. Every important external characteristic of the system should be present as an abstrac- at this stage. Many of the abstractions have to do with the management of system resources; others have do with services provided to the user.

The second phase of system design (phase 2) investigates the practicality of the abstractions proposed by phase 1 and establishes the data connections between the partitions and the flow of control among the parti- This latter exercise establishes the placement of the various partitions in the hierarchy. The second phase occurs concurrently with the first; as abstractions proposed, their utility and practicality are immediately investigated. For example, in an information retrieval system the question of whether a given search technique is efficient enough to satisfy system constraints must be investigated.

A partition has been adequately investigated when connections with the rest of the system are known when the designers are confident that they understand exactly what its effect on the system will be. V ying depths of analysis will be necessary to achieve this confidence. It may be necessary to analyze how the functions of the partition could be implemented, involving phase 1 analysis as new abstractions are postulated requiring lower partitions or sub-partitions. Possible results of a phase 2 investigation are that an abstraction be accepted with or without changes, or it may be rejected. If an abstraction is rejected, then another abstraction must be proposed (phase 1) and investigated (phase 2). The iteration between phase 1 and phase 2 continues until the design is complete.

Structured programming

It is not clear exactly how early structured programming of the system should begin. Obviously, whenever the urge is felt to draw a flowchart, a structured program should be written instead. Structured programs connecting all the partitions together will be expected by the end of the design phase. The best rule is probably to keep trying to write structured programs; failure will indicate that system abstractions are not yet sufficiently understood and perhaps this exercise will shed some light on where more effort is needed or where other abstractions are required.

When is the design finished?

The design will be considered finished when the following criteria are satisfied:

(1) All major abstractions have been identified and partitions defined for them; the system resources have been distributed among the partitions and their positions in the hierarchy established.

(2) The system exists as a structured program, showing how the flow of control passes among the partitions. The structured program consists of several components, but no component is likely to be completely defined; rather each component is likely to use the names of lower-level components which are not yet defined. The interfaces between the partitions have been defined and the relevant test cases for each partition have been identified.

(3) Sufficient information is available so that a skeleton of a user's guide to the system could be written. Many details of the guide would be filled in later, but new sections should not be needed.*

An example from Venus

The following example from the Venus system[4] is presented because it illustrates many of the points made about selection, implementation, and use of abstractions and partitions. The concept to be discussed is that of external segment name, referred to as ESN from now on.

The concept of ESN was introduced as an abstraction primarily for the benefit of users of the system. The important point is that a segment (named virtual memory) exists both conceptually (as a place where a

* This requirement helps to insure that the design fulfills the system specifications. In fact, if there is a customer for whom the system is being developed, a preliminary user's guide derived from the system design could be a means for reviewing and accepting the design.

pro[illegible]mmer thinks of information as being stored) and [illegible]y (the encoding of that information in the computer). The reality of a segment is supported by an internal segment name (ISN) which is not very convenient for a programmer to use or remember. Therefore, the symbolic ESN was introduced.

As soon as the concept of ESN was imagined, the existence of a partition supporting this concept was implied. This partition owned a nebulous data resource, a dictionary, which contained information about the mappings between ESNs and ISNs. The formatting of this data was hidden information as far as the rest of the system was concerned. In fact, decisions about the dictionary format and about the algorithms used to search a dictionary could safely be delayed until much later in the design process. A collective name, the dictionary functions, was given to the functions in this partition.

Now phase 2 analysis commenced. It was necessary to define the interface presented by the partition to the rest of the system. Obvious items of interest are ESNs and ISNs: the format of ISNs was already determined by the computer architecture, but it was necessary to decide about the format of ESNs. The most general format would be a count of the number of characters [illegible]he ESN followed by the ESN itself; for efficiency, [illegible]wever, a fixed format of six characters was selected.

At this point a generalization of the concept of ESN occurred, because it was recognized that a two-part ESN would be *more useful* than a single symbolic ESN. The first part of the ESN is the symbolic name of the dictionary which should be used to make the mapping; the second part is the symbolic name to be looked up in the dictionary. This concept was supported by the existence of a dictionary containing the names of all dictionaries. A format had to be chosen for telling dictionary functions which dictionary to use; for reasons of efficiency, the ISN of the dictionary was chosen (thus avoiding repeated conversions of dictionary ESN into dictionary ISN).

When phase 2 analysis was over, we had the identification of a partition; we knew what type of function belonged in this partition, what sort of interface it presented to the rest of the system, and what information was kept in dictionaries. As the system design proceeded, new dictionary functions were specified as needed. Two generalizations were realized later. The first was to add extra information to the dictionary; this was information which the system wanted on a segment basis, and the dictionaries were a handy place to [illegible]tore it. The second was to make use of dictionary functions as a general mapping device; for example, dictionaries are used to hold information about the mapping of record names into tape locations, providing simplification of a higher partition.

In reality, as soon as dictionaries and dictionary functions were conceived, a core of dictionary functions was implemented and tested. This is a common situation in building systems and did not cause any difficulty in this case. For one thing, extra space was purposely left in dictionary entries because we suspected we might want extra information there later although we did not then know what it was. The search algorithm selected was straight serial search; the search was embedded in two internal dictionary functions (a sub-partition) so that the format of the dictionaries might be changed and the search algorithm redefined with very little effect on the system or most of the dictionary functions. This follows the guideline of modifiability.

CONCLUSIONS

This paper has described a design methodology for the development of reliable software systems. The first part of the methodology is a definition of a "good" system modularization, in which the system is organized into a hierarchy of "partitions", each supporting an "abstraction" and having minimal connections with one another. The total system design, showing how control flows among the partitions, is expressed as a structured program, and thus the system structure is amenable to proof techniques.

The second part of the methodology addresses the question of how to achieve a system design having good modularity. The key to design is seen as the identification of "useful" abstractions which are introduced to help a designer think about the system; some methods of finding abstractions are suggested. Also included is a definition of the "end of design", at which time, in addition to having a system design with the desired structure, a preliminary user's guide to the system could be written as a way of checking that the system meets its specifications.

Although the methodology proposed in this paper is based on techniques which have contributed to the production of reliable software in the past, it is nevertheless largely intuitive, and may prove difficult to apply to real system design. The next step to be undertaken at MITRE is to test the methodology by conscientiously applying it, in conjunction with certain management techniques,[4] to the construction of a small, but complex, multi-user file management system. We hope that this exercise will lead to the refinement, extension and clarification of the methodology.

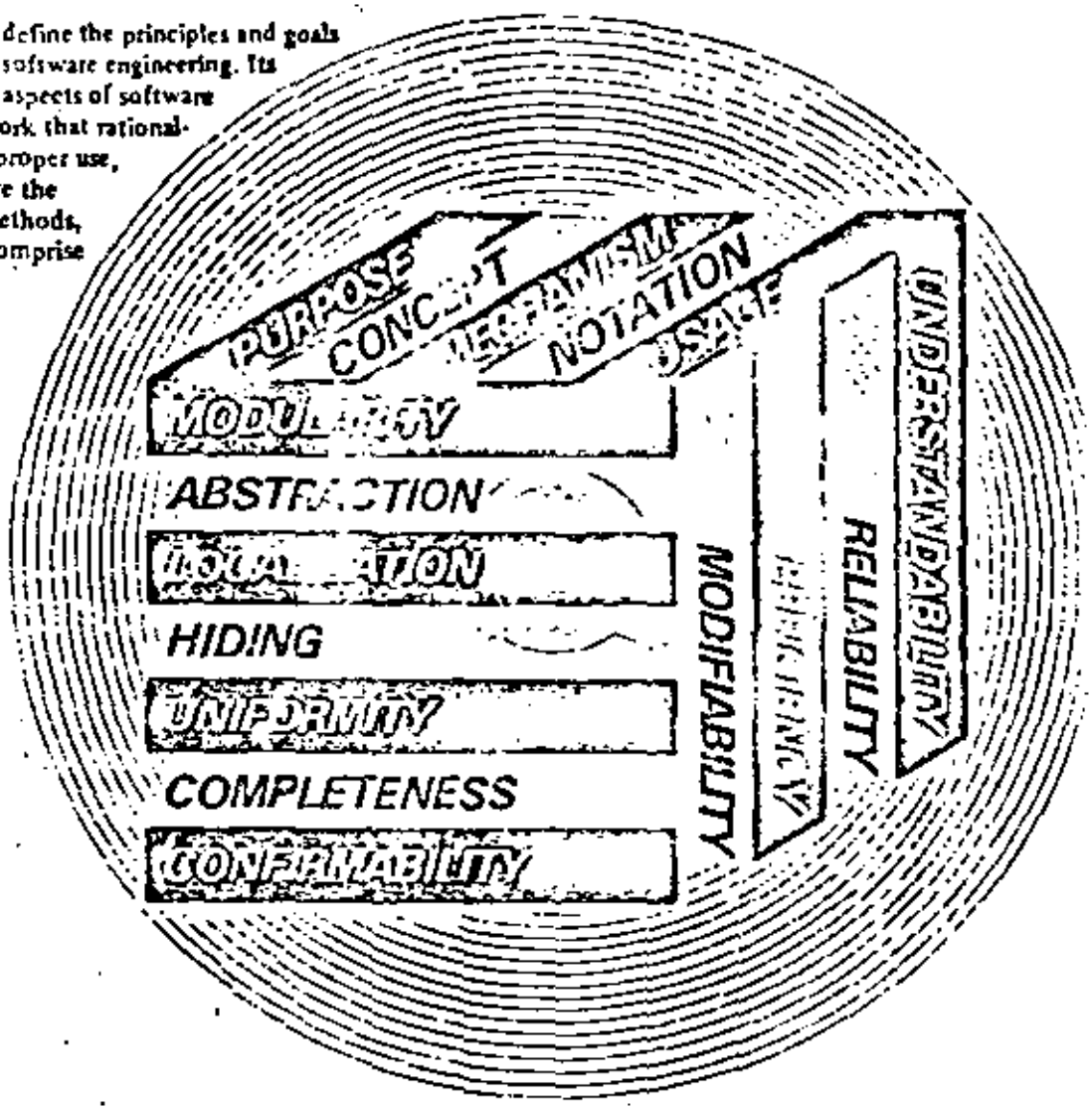
ACKNOWLEDGMENTS

The author wishes to thank J. A. Clapp and D. L. Parnas for many helpful criticisms.

REFERENCES

1 J N BUXTON B RANDELL (eds)
Software engineering techniques
Report on a Conference Sponsored by the NATO Science Committee Rome Italy p 20 1969
2 B H LISKOV E TOWSTER
The proof of correctness approach to reliable systems
The MITRE Corporation MTR 2073 Bedford Massachusetts 1971
3 W DIJKSTRA
Structured programming
Software Engineering Techniques
Report on a Conference sponsored by the NATO Science Committee Rome Italy J N Buxton and B Randell (eds) pp 84-88 1969
4 F T BAKER
Chief programmer team management of production programming
IBM Syst J 11 1 pp 56-73 1972
5 CONWAY
How do committees invent?
Datamation 14 4 pp 28-31 1968
6 B H LISKOV
Guidelines for the design and implementation of reliable software systems
The MITRE Corporation MTR 2345 Bedford Massachusetts 1972
7 D L PARNAS
Information distribution aspects of design methodology
Technical Report Department of Computer Science Carnegie-Mellon University 1971
8 E W DIJKSTRA
The structure of the "THE"—multiprogramming system
Comm ACM 11 5 pp 341-346 1968
9 S MADNICK J W ALSOP II
A modular approach to file system design
AFIPS Conference Proceedings 34 AFIPS Press Montvale New Jersey pp 1-13 1969
10 B H LISKOV
The design of the Venus operating system
Comm ACM 15 3 pp 144-149 1972
11 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven The Netherlands 1969
12 H D MILLS
Structured programming in large systems
Debugging Techniques in Large Systems R Rustin (ed) Prentice Hall Inc Englewood Cliffs New Jersey pp 41-55
13 P HENDERSON R SNOWDEN
An experiment in structured programming
BIT 12 pp 38-53 1972
14 D L PARNAS
On the criteria to be used in decomposing systems into modules
Technical Report CMU-CS-71-101 Carnegie-Mellon University 1971
15 A COHEN
Modular programs: Defining the module
Datamation 18 1 pp 34-37 1972

This paper attempts to define the principles and goals that affect the practice of software engineering. Its intent is to organize these aspects of software engineering into a framework that rationalizes and encourages their proper use, while placing in perspective the diversity of techniques, methods, and tools that presently comprise the subject of software engineering.



SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS

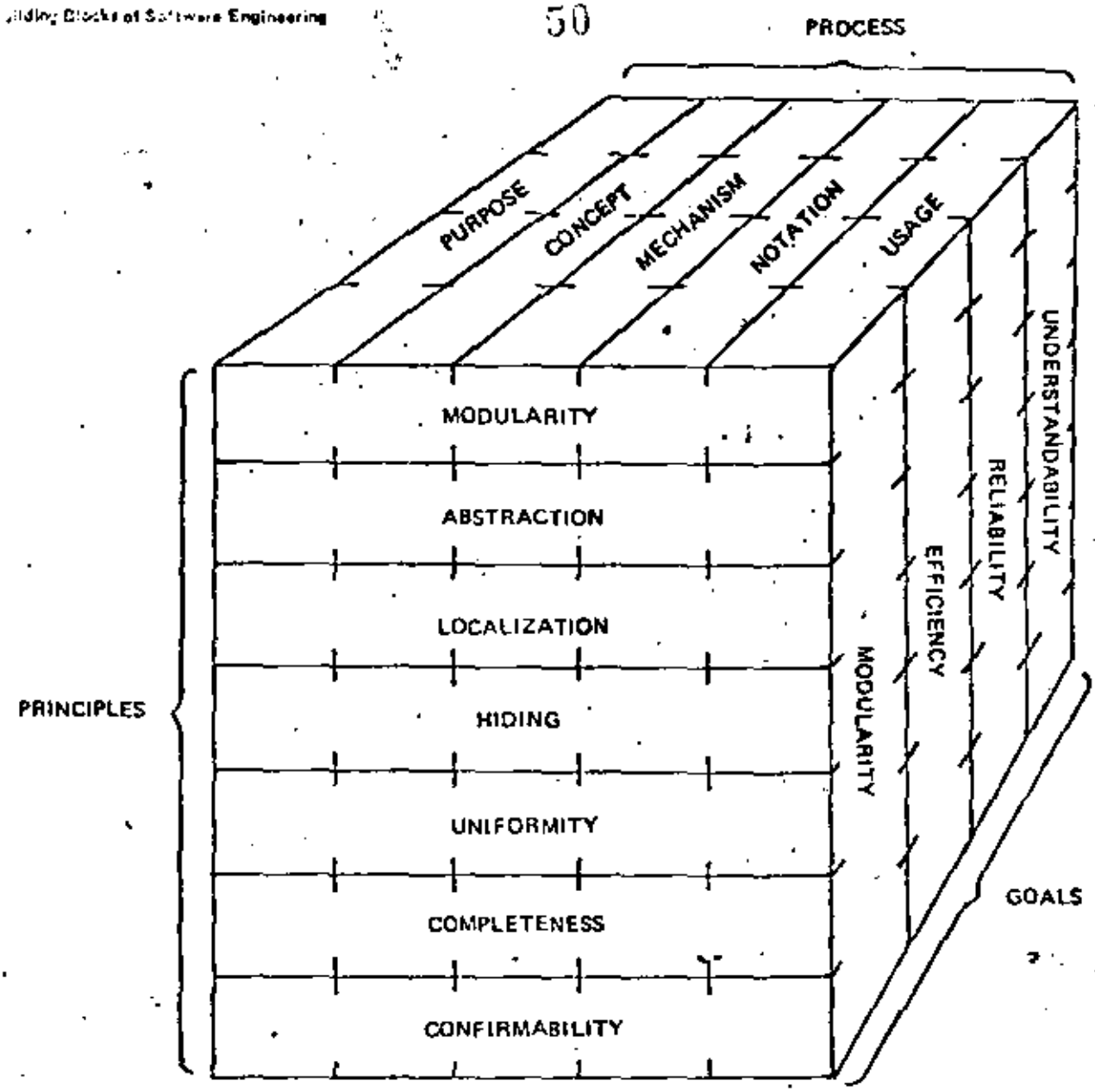
Douglas T. Ross, John B. Goodenough, C.A. Irvine
SofTech, Inc.

Introduction

The conferences sponsored by NATO in 1968 and 1969 gave popular impetus to the term "software engineering." Since that time the need for a more disciplined and integrated approach to software development has been increasingly recognized. Although useful definitions of the term remain elusive, software engineering clearly implies at least the disciplined and skillful use of suitable software development tools and methods, as well as a sound understanding of certain basic principles. In this paper, we attempt to expound what these principles are, and how they are applied in the practice of software engineering.

It is perhaps best to view this paper as an attempt to identify the important underlying issues of software engineering in a form that permits the interaction of these issues to be better understood. We will discuss these issues in terms of four fundamental goals: *modifiability*, *efficiency*, *reliability*, and *understandability* as well as seven principles that affect the process of attaining these goals:

- the *modularity* principle, which defines how to structure a software system appropriately;
- the *abstraction* principle, which helps to identify essential properties common to superficially different entities;
- the *hiding* principle, which highlights the importance of not merely abstracting common properties but of making inessential information *inaccessible* (hiding deals with defining and enforcing constraints on access to information);



- the *localization* principle, which highlights methods for bringing related things together into physical proximity;
- the *uniformity* principle, which ensures consistency;
- the *completeness* principle, which ensures that nothing is left out;
- the *confirmability* principle, which ensures that information needed to verify correctness has been explicitly stated.

The principles and goals are applied in the practice of software engineering, which deals with various software development activities:

Determine requirements—the process of identifying the requirements to be satisfied by a software system; the objective is to define the problem to be solved in terms of the constraints a solution must satisfy, including cost and performance.

Design software—the process of considering each user requirement and creating the conceptual basis on which the problem is to be solved; design is the process of deciding how to satisfy user requirements within the allowed constraints.

Specify implementation—the process of describing the interactions between the designed modules of a solution; the result of this activity is a detailed specification of constraints the software implementation must satisfy, but not the software itself.

Code/debug—the process of actually producing the software satisfying the specification, and verifying that the produced software does satisfy the user requirements.

Tuning—the process of modifying a logically correct system until it meets performance goals.

Despite the obvious differences among these activities, we believe each reflects a common pattern which we call the *fundamental process*. This process consists of five basic steps: (1) crystallize a *purpose* or objective; (2) formulate a *concept* for how the purpose can be achieved; (3) devise a *mechanism* that implements the conceptual structure; (4) introduce a *notation* for expressing the capabilities of the mechanism and invoking its use; (5) describe the *usage* of the notation in a specific problem context to invoke the mechanism so the purpose is achieved.

This sequence of steps has a natural parallel to the process of software development:

purpose – define the requirements for a system;

concept – derive the architecture of a software system to satisfy these requirements and specify the modules that constitute the system;

mechanism – implement the software system (devising the mechanism is obviously the code/debug/tune activities in the development process);

notation – define the command language or other means a user will employ to invoke the capabilities of the software system;

usage – describe how the software system is controlled (this description may take the form of a user's manual for the system).

Equally well, the pattern defined by the fundamental process could be applied differently, to highlight a different aspect of software development. For example, we could leave the description of purpose and concept as above, but replace mechanism, notation, and usage with the following descriptions:

mechanism – the computer on which the software runs;

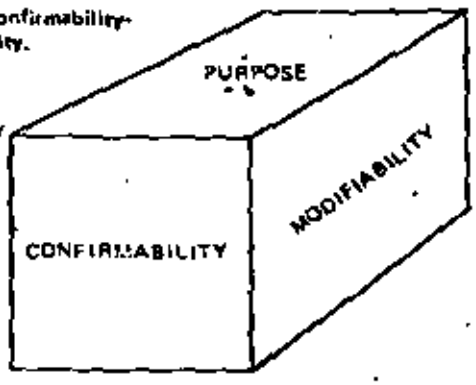
notation – the programming language in which the software will be written;

usage – the manual describing how the language is used to control the computer.

The interpretation of the fundamental process is clearly highly context-dependent. It is also intimately tied to the notion of *hierarchical decomposition*–the widely recognized phenomenon of part/whole relationships. A purpose is composed of sub-purposes; a mechanism has many parts which are themselves sub-mechanisms, and in general, the mechanism itself is only a part of some super-mechanism, etc. The interesting phenomenon is that *both* the pattern of the fundamental process *and* part/whole hierarchical relationships must be employed in all aspects of software engineering practice.

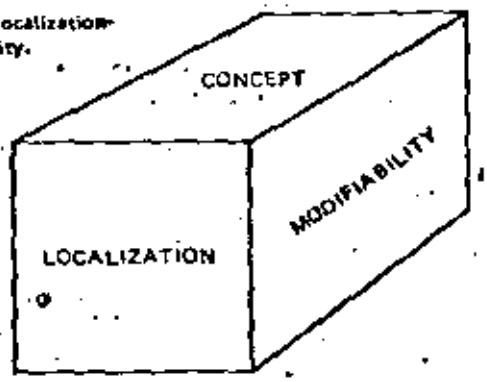
Given that part/whole hierarchical decomposition plays a pervasive role, the fundamental process interacts further with the various principles and goals of software engineering as depicted in Figure 1. Figure 1 is our framework for discussing the nature and issues of software engineering. Clearly an exhaustive treatment is not possible, but the remainder of this paper is intended to show how the structure of Figure 1 does yield insight into the theory and practice of software engineering. By way of illustration, consider the following examples (Figures 2-7) of how principles, goals, and process elements interact:

Figure 2. Purpose-Confirmability-Modifiability.



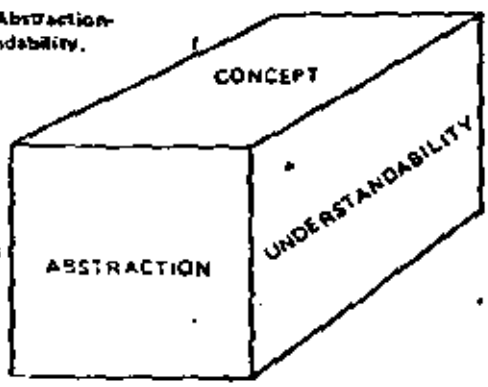
- The purpose of a design choice may be to structure a system so the effects of a change can be predicted with assurance.
- Confirmability applied to the process of defining purposes for modifiability demands that purposes be stated in a form that makes it possible to easily check if they have been achieved, e.g., an objective whose achievement would enhance modifiability would be to insure that only declarations in a program need be changed when transferring a program to a new computer. This is a confirmable statement of objectives while "reducing the number of changes that must be made" is not so clearly confirmable, and hence, is not so useful.

Figure 3. Concept-Localization-Modifiability.



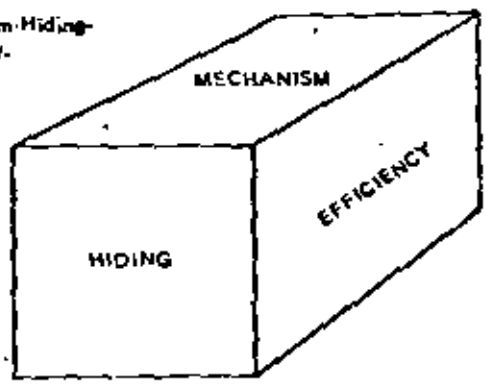
- Identifying one module for implementing a scheduling policy in an operating system is an example of the effect of the localization principle on design concepts to enhance modifiability, since localizing the policy rather than scattering policy decisions throughout a system makes it easier to change the policy.
- Having decided on the previous design for scheduling, the concept of a table-driven scheduler, which localizes scheduling policy in a single table *within* the module, then makes the *module* more modifiable. These two examples illustrate the role of hierarchy in applying the principles, goals, and process steps.

Figure 4. Concept-Abstraction-Understandability.



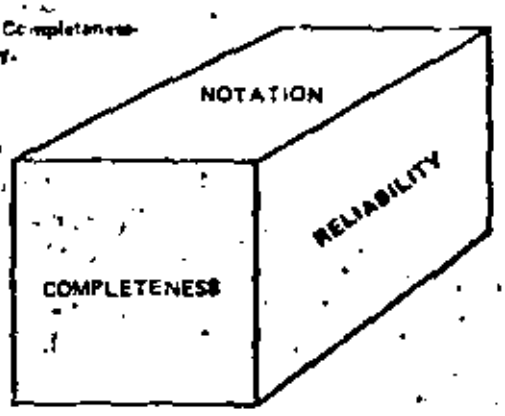
- The use of levels of abstraction[9,11,10] to define an understandable program or system of programs shows how abstraction can make designs more understandable.
- High-order languages represent a concept for improving the understandability of programs by abstracting from the details of computer instruction sets.

Figure 5. Mechanism-Hiding-Efficiency.



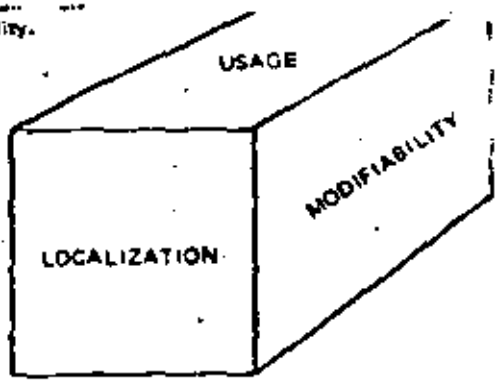
- Knowing that only certain subroutines have access to shared data (e.g., as in a function cluster[12]) may permit fewer checks to be made on the validity of stored values, and thereby increase efficiency.
- Forbidding users from directly accessing I/O devices permits an operating system to optimize overall usage of the devices.

Figure 6. Notation-Completeness-Reliability.



- Having a complete set of convenient goto-free control structures is necessary to avoid error-prone circumlocutions.
- In designing a case statement, if the form does not permit a programmer to specify what is to happen when the case statement variable is out of range, then the programmer is unable to easily treat this possibility, and hence is not encouraged to develop error recovery algorithms. A complete notation can foster reliability by permitting a programmer to specify error recovery details.

Figure 7. [illegible] Modifiability.



- An installation manual that is organized so that each user option and all installation-specific parameters are described in one place makes updating the manual easier when changes in these options and parameters occur.
- A cross-reference listing for a program localizes the description of how a particular variable is used, and thereby makes it easier to evaluate the effects of potential changes.

These examples can only serve to illustrate the style of approach (on a per-cube basis) used to make the goals, principles, and parts of the fundamental process useful in describing and understanding aspects of software engineering. The following sections discuss the hierarchy, goals, and principles in more specific terms before we apply them jointly in an extended example.

Hierarchical Decomposition

The process of developing hierarchical structure may be viewed *top-down* as successively imposing increasingly specific constraints on the form of an ultimate solution. It may also be viewed *bottom-up* as successively exploiting the constraints satisfied by lower levels. An understanding of software engineering lies in understanding the *constraints* each engineering activity places on the others—e.g., how the design constrains the implementation, and how the implementation possibilities constrain the design. The process of hierarchically and iteratively imposing constraints appropriate to the analysis, design, specification, and coding phases is what unifies the practice of software engineering.

The process of hierarchical decomposition involves both analysis and synthesis—both taking apart and putting together. While one decomposes a subject by recognizing the *different* sub-parts of which it is composed, one must also pause to examine the overall decomposition and look for *similarities* among the lower-level constituents with an eye toward recomposing collections of them into larger constructs. For example, pure decomposition would never result in recognizing subroutines that are needed in separated parts of the decomposition.

For a given problem there are many "correct" decompositions, and given a large collection of solutions to primitive sub-problems (e.g., a set of CPU instructions), there are many correct compositions which will solve a given problem. Thus, whether one is engaged in synthesis or analysis, composition or decomposition, the selection of a "correct" solution must be based upon some overriding criteria. We must try to select the most desirable solution from the set of correct solutions. For this reason we begin first, in our elaboration of the thesis of this paper, by considering the goals of software engineering.

The Goals of Software Engineering

The skill with which we can apply engineering methods and tools will depend on the degree to which we have a clear and precise view of our objectives. In the realm of software engineering our objectives will always be stated in terms of desired properties of the resultant software. Four properties that are sufficiently general to be accepted as *goals* for the entire discipline of software engineering are *modifiability*, *efficiency*, *reliability*, and *understandability*.

The goal of *modifiability* is historically the most difficult goal to master. Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained. The characterization of "sameness" or invariance may be very subtle, and the effects of change may be hard to predict. This makes the achievement of modifiability difficult. Modifiability is also difficult to achieve because changes occur for so many different types of reasons. For example, in transferring software to a new computer or operating system, it is desired to keep invariant the logical effects of the system, limiting changes only to the necessarily machine-dependent aspects. Changes are also required to remove errors from software, to add new capabilities, and to improve a system's performance. In general, different approaches are necessary to satisfy these different types of modifiability.

Modifiability requires not only the ability to have an adaptable, evolutionary design, employ standardized software building-blocks, tune for performance, etc., but also the more subtle ability to maintain project schedules and budgets by allowing dummy test modules to be used as drivers before later parts of a system are prepared, etc. The variety of ways modifiability affects software engineering is one of the reasons for giving it a primary role in our discussion of software engineering.

A much-abused goal is *efficiency*, usually because in an excess of zeal it is prematurely permitted a high priority in engineering tradeoffs. Blatant *inefficiency* cannot, of course, be tolerated, but usually efficiency questions are best treated within the context of other issues. For example, achieving a high degree of modifiability can provide the basis for meeting efficiency goals during the tuning phase of software development. In addition, insights

reflecting a more unified understanding of a problem have far more impact on efficiency (via abstraction and uniformity) than any amount of bit twiddling within a faulty structure. In general, except for the highest ptual levels of a design, where gross inefficiency q. ions may play dominant roles, the efficiency goal does not dominate the practice of software engineering.

Reliability is a goal much in vogue today. Reliability must both prevent failure in conception, design, and constructi.t, as well as recover from failure in operation or performance. Unlike efficiency, which frequently is prematurely applied, reliability is more often considered too late, or not at all, in most software development efforts. Reliability can only be built in from the start; it cannot be added on at the end. Hence, reliability has a pervasive and crucial effect on software engineering practices.

The final goal which should exert a strong influence in all aspects of software engineering is *understandability*. It depends, of course, on the intended audience: technical, management, or user. Note in particular that understandability is not merely a property of legibility. Much more importantly, the entire conceptual structure is involved. Also, in any given circumstance an acceptable level of understandability either is or is not present. There is no middle ground. Although understandability is, in a sense, a prerequisite to reliability and modifiability, it is also important as a goal in itself because it draws attention to an important barrier to understandability—complexity. Management of complexity is a crucial aspect of software engineering methods, and the need to manage complexity i from the goal of understandability. The only way to . .ieve the goal of understandability with regard to an inherently complex system is to impose an appropriate structure and organization on the system. The structure must be represented in a clear notation that permits mechanical translators—e.g., compilers, etc.—to bridge the gap between the actual system and an understandable representation of it. Thus, achieving understandability depends as much upon the software engineering tools as on the methods. For example, when compilers do not produce acceptably efficient code, assembly language may have to be used, at a cost in program understandability.

The Principles of Software Engineering

The principles of software engineering are, as we mentioned earlier, *modularity*, *abstraction*, *localization*, *hiding*, *uniformity*, *completeness*, and *confirmability*. These *principles* applied in various combinations within the *fundamental process* will work to produce *hierarchical decompositions* which achieve our *goals* during all of the various phases of software development. The hierarchical decomposition of a system depicts the constituents of the system organized into a structure by the relationships among those constituents. The above seven principles, and in combination, are used to determine and ol those relationships. They are used essentially as decision criteria to ensure that the resulting decomposition attains our goals, and thus each deals with some aspect of the relationships—i.e., the interfaces among the constituents.

Modularity Modularity deals with properties of hierarchical software structures. It has been given various definitions. Sometimes it has been defined in terms of objectives:

> "[One objective of modular programming is to be able to convince oneself] of the correctness of a program module, independently of the context of its use in building larger units of software." (Reference 2, p. 129)

> "Modularity denotes the ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules." (Reference 9, p. 54)

> "Modularity is not . . . simply the arbitrary division of a large program into smaller parts or modules. The primary goal . . . should be to decompose the program in such a way that the modules are highly independent from one another." (Reference 13, p. 100)

Modularity is more frequently defined in terms of structural properties possessed by "modular" systems:

> "A program is modular if it is written in many relatively independent parts or modules which have well-defined interfaces such that each module makes no assumptions about the operation of other modules except what is contained in the interface specifications." (Reference 4, p. 1)

> "Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. . . . A definition of "good" modularity must emphasize the requirement that modules be as disjoint as possible." (Reference 11, p. 192)

> "A modular program is a program [having a hierarchical structure]. (Reference 1, p. 34)

> "Modular programming is the organizing of a complete program into a number of small units . . . where there is a set of rules which controls the characteristics of those units. (Cited in Reference 10, p. 29)

In general, modularity is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. Most discussions of modularity focus on one or another of these objectives and attempt to explain why certain structural constraints make the attainment of these objectives easier.

Rather than select any one objective as the most important one, we propose[5] a general unifying definition:

> Modularity deals with how the *structure* of an object can make the attainment of some *purpose* easier. Modularity is *purposeful structuring*.

Hence, the principle of modularity is made concrete by explaining how certain constraints on the structure of systems can make it easier or harder to achieve some purpose.

For example, what sort of structural constraints facilitate modifiability? efficiency? reliability? Imposing such constraints on structures is the essence of applying the modularity principle in software engineering.[6,7] For example, goto-free programming forces programmers to make explicit the conditions under which a given statement is executed, and this can help ensure understandability and prevent errors.

The principles of modularity can be further illustrated by considering modularity issues arising in deciding what parts belong in which ships should be considered in developing hierarchical decompositions:

Increasing Machine Dependence The lower the module in the system structure, the more the module is dependent on the hardware on which it runs. This helps to make software more portable, by isolating machine-dependencies.

Refinement of action Higher level modules specify objectives for some action, i.e., *what* is to be done. Lower levels describe how the objective is going to be realized, i.e., *how* something is to be done. For example, within a higher level module, an engineer might specify that a temperature is to be adjusted until it is at least 145°. A *lower level* module will define how this adjustment is performed, e.g., by adjusting and sampling the temperature trend at five minute intervals. This constraint helps make a system more understandable and easier to modify.

Scope of control Higher level modules call lower level modules and supervise their activities. This means that if lower level modules encounter some exception condition (e.g., a condition that prevents the requested operation from being performed), this must be reported to higher level modules so they can take appropriate action.

It may be impossible for a given program to satisfy all these objectives simultaneously. A program may have one structure if modules are ordered according to one rule, and a different structure if a different rule is considered. The main point in identifying these relationships is to highlight possible criteria that can be consciously used in structuring the modules. Many of these criteria are already used instinctively; the advantage of making the criteria explicit is that any programmer produces better results if he is consciously aware of criteria for evaluating what he is doing.

Abstraction Like modularity, *abstraction* is a very pervasive principle. The essence of abstraction is to extract essential properties while omitting inessential details. Our discussion of hierarchical decomposition in the form of "levels" showed abstraction in perhaps its most pristine form. Each level of the decomposition presents an abstract view of the lower levels purely in the sense that details are subordinated to the lower levels.

The principle of abstraction when combined with the principle of completeness ensures that a given level in a decomposition is understandable as a unit, without requiring either knowledge of lower levels of detail, or necessarily how it participates in the system as viewed from a higher level. Thus this principle is employed on the one hand to obtain a description of some level of the system which could be realized by any of several implementations, and on the other hand to give a description of one part of a system which could be used in many other systems requiring the same component at that level of abstraction.

The principle of abstraction interacts very strongly with the purpose underlying any particular decomposition. The principle is of little practical value unless combined with the principle of modularity ("purposeful structuring") to ensure that appropriate abstractions are found. Abstractions employed to achieve the goal of understandability must fit each level of abstraction, while preventing more and more detailed views of the system, must do so in terms which are understandable to the intended audience.

Localization The principle of localization is concerned with physical proximity. Things which are thought together all in one place. Thus, localization deals with physical interfaces, textual sequence, memory, etc. Then the only principles can interrelate the localized things to serve particular purposes. Consider carefully how intimate is the connection between localization of an abstraction in a module and our ability then to understand and deal with it.

Subroutines, arrays, logical and physical records, as well as paged memories, are examples of localization. The avoidance of goto's in structured programming is an application of localization to control structures which enhances understandability and simplifies confirmability.

Hiding Another principle familiar to many readers is *hiding*. Parnas,[13] for example, uses it as the major criterion for a decomposition into modules. It is related to the idea of "postponing binding decisions" in top-down problem-solving, although it is not the same. The purpose of hiding is similar to that of abstraction in that it requires making visible only those properties of a module needed to interface with other modules. But hiding differs from abstraction in that the purpose of hiding is *to make inaccessible* certain details that should not affect other parts of a system. Abstraction helps to identify details that should be hidden. Hiding is concerned with *defining and enforcing access constraints* that, without the hiding principle, would otherwise only be implicit in some purpose, concept, mechanism, notation, or usage description.

Hiding, combined with abstraction and localization, forces suppression of *how* to emphasize *what*. Suppressing how a constraint is satisfied forces the constraint to be made more explicit, thereby amplifying our understanding of the constraint itself. Deciding what constraints are to be expressed (an aspect of modularity—purposeful constraint selection) is a matter independent of the hiding principle itself.

Uniformity Uniformity (the lack of inconsistencies and unnecessary differences) is also an important principle. When applied to notational matters, uniformity yields a notation free of confusing and perhaps costly inconsistencies. When also combined with the abstraction principle, uniformity implies a notation that permits arbitrary mechanization of the internal detailing of a mechanism—the notation does not constrain one's choice of implementation. And when the hiding principle is added, the result is a notation that does not merely permit several implementation choices, but also ensures that no unnecessary details of a specific implementation are revealed by the notation. For example, if a subroutine parameter is to be a stack, the representation of the stack should usually be hidden from the user of the subroutine. Thus, the user should be able to allocate storage for stacks and pass them to subroutines without having access to the individual components of a stack. A notation for representing such abstract data types is given in References 7 and 12. Another example is given in a recent paper on exception handling methods,[9] in which a uniform notation is proposed whose semantics can be realized using various traditional methods of handling error returns from subroutines.

In its essence, notation satisfying the uniformity and abstraction concepts has been called elsewhere[10] the

Uniform Referent Principle. This principle, applied to the concept formation step of the fundamental process, yields a consistent and well-defined set of semantic concepts that can be represented with a uniform syntax. A uniform notational syntax is not possible if there is no semantic uniformity underlying the notation.

55

Completeness Completeness is obviously an important principle. The purpose of this principle is to ensure that all the essentials of an abstraction, for example, are explicit and that nothing essential has been omitted. This does not require that every *detail* be shown—merely that the set of abstract concepts *covers* every detail. Applied to notational matters, completeness requires that a notation provides a means for saying everything that one wants to say. Combined with abstraction, it implies that a notation should be concise, permitting the suppression of invariant details in favor of highlighting the potentially changeable. Completeness combined with uniformity and abstraction and applied to the goal of efficiency suggests that programmers should be able to select different implementation mechanisms to tune a system's performance, but without changing the form of any subroutine call, for example. A notation that does not permit this purpose to be achieved is incomplete.

Confirmability Confirmability is a principle that directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or implementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walk-throughs of designs, egoless programming,[18] and other methods that help to ensure that nothing has been overlooked.

The principle of confirmability can be realized in many useful forms, both as entirely manual procedures and strongly aided by the tools and data base of a software engineering facility. Certain kinds of type checking and consistency checking reflect the principle of confirmability applied to the design of programming languages and compilers.

Completeness and confirmability are easily confused. For example, in the Introduction, we presented examples illustrating the interaction between notation, completeness, and reliability. In one of these examples, we noted that to ensure completeness of case statement control a programmer should be *permitted* by the syntax to specify what should happen when a case statement variable is out of range. Confirmability applied to the same issue would imply a programmer should be *required* to state what should happen. Of course, if he knows that out-of-range values are not possible, this too should be expressible, to [illegible] implementation efficiency. In short, the evolution of completeness to satisfy confirmability requires that otherwise obscure implications be made to show in explicit form.

An Example of the Framework's Utility

Having discussed the components of the process/goal/principle framework at greater length, we now intend to show how the framework can be used to gain and structure insights into aspects of software engineering. By giving an extended example, we hope to show that the framework is not merely taxonomic but can actively assist in dealing with the complexities of software engineering.

Our example will show how the framework can help to organize our understanding of the notion of a subroutine. We choose this example because, although "subroutine" is fundamental to software, and seems well-understood, it is also one of the most complex concepts when considered in its totality. Figure 8 shows the pattern of the fundamental process applied to the subroutine concept. The notation proposed is essentially that of ALGOL 60. Other notations could have been proposed equally well. The description of the subroutine concept in Figure 8 is obviously very general. In particular, the description of "Mechanism" is at a high level of abstraction.

Applying hierarchical decomposition, the mechanism aspect of subroutines can be decomposed into two less-abstract mechanisms for implementing the subroutine concept—one for inline subroutines and one for closed subroutines. Then, the fundamental process can be applied again with respect to these mechanisms, as shown in Figures 9 and 10. We have inserted parenthetical comments to show how various principles and/or goals are being served.

Consider now the closed subroutine, Figure 9. Our description holds no surprises, for we are still at a high level. The notions of Figure 9 could be refined in several ways, however. For example, there are at least two distinct kinds of subroutine linkage mechanisms: 1) *direct linkage*, which is usually supported by a machine instruction that saves the return address and transfers control to the subroutine body, and 2) *indirect linkage*, a mechanism employed in AED implementations,[17] in which a subroutine call is implemented not directly by transferring control to the subroutine, but indirectly, by transferring control to a "linkage" subroutine. The purpose of this is to save space on machines for which stack manipulations are expensive and to *localize* at run-time all subroutine calls so different linkage routines can be substituted—e.g., a timing linkage that gathers information about how much time is spent in each subroutine, or a debugging linkage that permits interception and tracing of calls by a debugging package. This application of the localization principle to subroutine linkage has the advantage of making it easier to satisfy the efficiency goal (by gathering timing information important in tuning a system) and the reliability goal (by making it easier to track down bugs). Furthermore, when complex calling sequences are required by the language implementation, the slight run-time cost of enter and leave macro operations yields significant storage savings through localization and sharing of the machine instructions needed for each call.

The call-return *concept* could also be explicated further by discussing more specific examples of the concept, in the style of Figure 9. For example, the use of stack frames (e.g., see Reference 14) vs. the storing of return addresses and other information related to subroutine invocation in each subroutine's storage space can be clarified by

Purpose: To invoke a similar pattern of action from many places in a program.

Concept: "Similar" but not "the same" implies that the "same" parts must be collected in one place, but combined in each case with the "different" parts. The "combination" must show how the "different" and "same" parts interact.

Mechanism: The "same" part is a subroutine body, located in one place and referenced by name. The "different" parts are the actual parameter values associated in each case with the subroutine name. The parameter-passing mechanism and coding of the subroutine body implement the "combination" of the parameters with the body at call time.

Notation:

<procedure statement> ::= <procedure identifier> [([<actual parameter>,...])]

<actual parameter> ::= { <string> | <expression> | <array identifier> | <switch identifier> | <procedure identifier> }

<procedure declaration> ::= [<type>] procedure <identifier> [([<identifier>,...])]; →
→ [value <identifier>,...;] →
→ [<specifier> <identifier>,...;] — →
→ <statement>

Usage: The syntax and semantics of a programming language define the contexts in which procedures can be invoked.

Figure 8. Top Level Explication of the Subroutine Call Concept

explicitly expressing purpose, concept, mechanism, notation, and usage."

It is useful to note the difference in the "Purpose" components of Figures 9 and 10. Although the goals inherited by decomposition from Figure 8 are the same—improve efficiency—note in Figure 10 that inline subroutines can improve efficiency in two ways: 1) by eliminating subroutine call overhead and 2) by performing certain computations at compile-time rather than at run-time, using actual parameter values. For example, if all parameters are constants, it may be possible to compute the value of the subroutine at compile time with consequent enormous savings in run-time efficiency. Wegbreit[19] explores this possibility and related ones in more detail.

For purposes of illustrating the use of our proposed framework, however, it is worth noting how Figures 9 and 10 help in comparing and contrasting two related but differing interpretations (inline versus closed) of the basic

Purpose:
- To save space by executing the same body of code with different parameter values

Concept:
- Call-return capability (transfer control to the subroutine's body, remembering where the call came from)

Mechanism:
- Specific calling sequences, e.g.:
  - direct linkage
  - indirect linkage

Notation:
- Notation for call should not be different from notation used to invoke inline subroutines. (Note that this is an application of the uniformity principle, and serves to foster program modifiability.)

Usage:

Figure 9. Description of the Closed Subroutine Concept

Purpose:
- To save time by eliminating call overhead (the efficiency goal);
- To save execution time by permitting compile-time simplification of a subroutine body based on actual values of parameters

Concept:
- The subroutine body is substituted in place of the call, with actual parameter values substituted for formal parameter values

Mechanism:
- Macro substitution, followed by compile-time optimization; or
- Syntactic substitution, in the sense that local variables declared within the subroutine will not be found to conflict with similarly-named variables in the context of the call, as might happen with macro substitution (which occurs at the lexical level of a program text).

Notation:
- Should not be different from call notation used to invoke closed subroutines. (This is an application of the uniformity principle, and serves to foster program modifiability.)

Usage:

Figure 10. Description of the Inline Subroutine Concept

subroutine notion. Note also how we have applied the framework (using hierarchical decomposition) to alternative "Mechanism" and to alternative "Concept" components of patterns in Figures 9 and 10. This recursive application of the pattern within components of the pattern is a principal attraction of using the framework for understanding software engineering topics in depth.

Our illustration so far appears to imply that the process aspect of the framework is of paramount importance, because we have organized our examples primarily in terms of this pattern. But each of the components of the framework is of equal importance, so an analysis can be organized equally well in terms of goals or principles. Depending on which dimension is used, the emphasis will be different, but using any of the three components as the dominating organizing concept is valid in applying the framework. Which of them should be used depends on the purpose of the discussion.

To demonstrate the validity and value of using one of the other components as the organizing principle, we describe in Figure 11 the notation aspect of subroutines, organized in terms of principles satisfied by various subroutine call notations. We will discuss each briefly below.

The purpose of *confirmability* as applied to notational matters is to ensure that important properties of a

Confirmability:

Purpose:	To ensure errors in forming a call are detectable.
Concept:	Distinguish input and output parameters.
Mechanism:	Use a colon or a semicolon to separate input and output parameters, e.g., F(A,B:C,D) or F(A,B;C,D).
Concept:	Provide indication of what exceptions can be raised.
Mechanism:	See below, under Completeness.

Conformity:

Purpose:	Avoid unnecessary differences in form of call.
Concept:	Ensure closed and inline calls have the same notational form. (This enhances modifiability directly, and efficiency indirectly.)

Completeness:

Purpose:	To ensure all properties of subroutines are reflected in the notation.
Concept:	Parameters should be both readable and writeable; notations for expressing response to exceptions should be available for use.
Mechanism:	• For read/write parameters: Use punctuation to separate input and output parameters. Declare which parameters are input and which are output. Incorporate body of subroutine in program where it is referenced so global analysis can determine which parameters are input and which are output (e.g., as in ALGOL). • Notations to deal with the various types of exception conditions.

Localization:

Purpose:	To express only once otherwise redundant information concerning exceptions.
Concept:	Associate handler with larger syntactic unit than the call itself.
Mechanism:	Use notation suggested in Reference 8.

Hiding:

Purpose:	To prohibit access to information that should be available only to the subroutine.
Concept:	Use abstract data type in declaring an actual parameter's data type, but a more detailed declaration of the formal parameter's type.

Abstraction:

Purpose:	To preserve essential properties of call in the notation, leaving other details to other notational devices.
Concept:	The method for handling exceptions should not be closely linked to implementation methods; a choice of a variety of implementation techniques should not be foreclosed by the notation.
Mechanism:	Use an implementation-neutral notation for exception handling.

Modularity:

Purpose:	To ensure that syntactic structure fosters appropriate goals.
Concept:	Examine impact of syntax on goal achievement.
Mechanism:	Choose the JOVIAL method of distinguishing input/output parameters rather than a declarative method, since the JOVIAL notation improves understandability.

Figure 11. Applying the Framework to Subroutine Call Notation Issues

subroutine's interface are explicitly in a call, it clear whether they have all been dealt with correctly. Two aspects of a call deserve particular mention as concepts for achieving this purpose. The first is to distinguish in the notation of the call which parameters are read-only (i.e., which are input parameters) and which are writeable (i.e., output parameters). The second is to distinguish in the form of the call what exception conditions a subroutine can raise.

PL/I is deficient in that no indication of output parameters is made. In this respect JOVIAL is superior, since a call to a JOVIAL subroutine, e.g., F(A,B:C,D), shows explicitly that A and B are input parameters and C and D are output parameters. In this case, the JOVIAL syntax is an example of a notational mechanism for implementing this concept. An equally good *mechanism* (considered solely from a confirmability viewpoint) would be merely to require that in the declaration of a subroutine, the input/output attributes of parameters be specified explicitly, so the requirement can be checked at compile-time. Also, uniformity with more modern programming languages, as well as ordinary English, might say that the ":" of JOVIAL might better be a "," to further improve the understandability of its syntax.

The explicit indication of exception conditions is a confirmability issue in that it permits oversights with respect to exception conditions to be detected more easily. We will discuss this concept further under Completeness.

As for *uniformity*, we list merely the concept that inline and closed subroutine invocations should have the same form. This enhances modifiability for tuning (efficiency) purposes, since changing a decision about whether to treat a subroutine as closed or inline will not then require changing every call.

Under *completeness*, we again list the concept that a subroutine's invocation notation should *provide* for dealing with exception conditions. The purpose served here is not to ensure that all conditions are dealt with appropriately (as for confirmability) but rather to ensure that every capability of the subroutine concept is mapped into a suitable notation for invoking that capability. The ability to control the response to an exception is an important aspect of subroutine invocation, and notation should be provided to deal with it. The confirmability purpose perhaps provides a stronger argument for associating exception handling with calls, but we cite it here because it also satisfies the completeness principle as well. Similarly, introducing the ability to assign to parameters is a concept suggested by the completeness principle; distinguishing input and output parameters in the form of the call or in a declaration is an application of the confirmability principle, because this make it easier to detect errors at compile time.

The purpose of *localization* as applied to notational matters for dealing with exception conditions might be stated as, "When the same exception handler is to be associated with several calling points for the same subroutine, the notation for exception handling should permit the handler to be written once, rather than requiring it to be written as part of each call." One concept for satisfying this purpose is to associate handlers with larger syntactic units of text than the call itself—e.g., with statements, loop bodies, etc. This proposal is explored further in Reference 8 and a specific syntax is proposed there.

[illegible] means allowing the subroutine access *only* to the abstract data type information needed for the semantics of the call. Thus declarations of formal parameters (i.e., those inside of the subroutine) are broken into two parts, and only the abstract part is *made* available to the user for his declarations (e.g., see Reference 12).

The principle of *abstraction* combined with uniformity suggests that the notation for dealing with exception conditions should be *neutral* with respect to various implementation techniques for handling exceptions. In Reference 8, a notation for exception handling is proposed that can be implemented using status variables, return codes, subroutines passed as parameters, or PL/I ON conditions as implementation methods for dealing with exceptions, depending on the logical constraints associated with the exception. The point is that the notation should permit a programmer to deal with the abstract concept of an exception, without being tied down to implementation details until he is ready to tune a system. This point is explored in greater detail in the cited reference.

Finally, applying the *modularity* principle to notational matters means exploring how the structural constraints imposed by a syntax can help to achieve some purpose. For example, the goal of understandability is enhanced by JOVIAL's syntax for distinguishing input and output parameters, as opposed to declaring which parameters are input parameters but not distinguishing in the structure of the call which parameter is an input parameter. Of course, the JOVIAL notation degrades modifiability, in that should an input parameter ever be changed to an output parameter, all calls would have to be modified, whereas the localization inherent in a separate declaration of read/write properties would not have this drawback. Human judgement is used to make a tradeoff decision in this case. The point is that by considering the effect of syntactic structure on achieving some goal, we have shown how the modularity principle applies to notational issues.

In short, Figure 11 shows that it is equally possible to organize a discussion of aspects of the subroutine concept in terms of principles as in terms of the fundamental process/pattern. (As an exercise, the reader might consider what principles are satisfied or degraded by the notational concept of optional arguments.)

The analysis in Figures 8, 9, 10, and 11 is only the beginning of a complete explication of the subroutine concept, but we hope our discussion has shown that by recursively applying the framework, in conjunction with hierarchical decomposition, organized and insightful explications can be developed to account for the virtues and deficiencies of various software engineering purposes, concepts, mechanisms, notations, and usages.

Conclusion

Our intent in this paper has been to consolidate and structure software engineering ideas into a coherent and useful framework for understanding the role these ideas play. The principles, goals, and process steps comprising this framework are not our inventions; they have been recognized by careful observers of software engineering for many years. We have merely attempted to present these ideas in an orderly and well-defined way. We do not claim to have identified all the important principles or goals

relevant to software engineering, but we are sure of the importance of the ones we have identified. We hope that others will find this approach useful in both understanding and improving the software engineering state of the art. ■

References

1. A. Cohen, "Modular Programs: Defining the Module," *Datamation*, vol. 18, pp. 34-37, January 1972.

2. J. B. Dennis, "Modularity," in *Advanced Course on Software Engineering*, F. L. Bauer, Ed. New York: Springer-Verlag, 1973, pp. 128-182.

3. E. W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, New York: Academic Press, 1972, pp. 1-82.

4. J. Earley, "Naming Structure and Modularity in Programming Languages," University of California, Berkeley, Calif., Tech. Rpt. TR-17, August 1973.

5. J. B. Goodenough and D. T. Ross, "System Organization Methodology: an Analysis of Modularity," in *MAIDS Information Dynamics Technology Requirements Study*, NTIS Doc. AD 768 898/9, June 1973.

6. J. B. Goodenough and D. T. Ross, "The Effect of Software Structure on Software Reliability, Modifiability, Reusability, and Efficiency: a Preliminary Analysis," SofTech, Inc., Waltham, Mass., NTIS Doc. AD 780 841, July 1973.

7. J. B. Goodenough and R. Zara, "The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: a Case Study and Analysis," SofTech, Inc., Waltham, Mass., NTIS Doc. AD 787 307/8, July 1974.

8. J. B. Goodenough, "Structured Exception Handling," *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pp. 204-224, January 1975.

9. G. Goos, "Language Characteristics: Programming Languages as a Tool in Writing System Software," in *Advanced Course on Software Engineering*, F. L. Bauer, Ed., New York: Springer-Verlag, 1973, pp. 47-69.

10. R. M. Gordon, "Implications of Using Modular Programming," Review of Central Computer Agency Guide No. 1, Civil Service Dept., Her Majesty's Stationery Office, London, England, *Datamation*, vol. 20, p. 29, November 1974.

11. B. H. Liskov, "A Design Methodology for Reliable Software Systems," *Proc. 1972 FJCC*, pp. 191-199.

12. B. H. Liskov and S. Zilles, "An Approach to Abstraction," *SIGPLAN Notices*, vol. 9, pp. 50-59, April 1974.

13. G. J. Myers, "Characteristics of Composite Design," *Datamation*, vol. 19, pp. 100-102, September 1973.

14. E. I. Organick, *The MULTICS System: An Examination of Its Structure*, Cambridge, Mass.: MIT Press, 1972.

15. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *CACM*, Vol. 15, pp. 1053-1058, December 1972.

16. D. T. Ross, "Uniform Referents: an Essential Property for a Software Engineering Language," in *Software Engineering*, J. T. Tou, Ed., Vol. 1, New York: Academic Press, 1970, pp. 91-101.

17. SofTech, "AED User's Guide," Tech. Rpt. G-AED-003-2, SofTech, Inc., Waltham, Mass., November 1972.

18. B. Wegbreit, "Procedure Closure in ELL," *The Computer Journal*, Vol. 17, pp. 38-43, January 1974.

19. G. M. Weinberg, *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971.

20. M. Woodger, "On Semantic Levels in Programming," *Proc. IFIP Congress 71*, pp. 402-407.

Douglas T. Ross is President and founder of SofTech, as well as its chief technical leader. An internationally known authority on software, he is the creator of the APT System for automatic programming of numerically-controlled machine tools; the AED Language and Programming System and the Algorithmic Theory of Language on which it is based; Structured Analysis, a new technique bringing the disciplined approach of software engineering to the systems analysis area; and the Plex Theory of problem modeling which provides a common theoretical base for his other achievements.

Prior to founding SofTech in July, 1969, Ross was on the Research Staff of the Electronic Systems Laboratory at MIT, serving as Head of the Computer Applications Group from 1956 onward. He was an organizer and participant in the NATO Software Engineering Conference in Garmisch, Germany (1968) and Rome, Italy (1969). His papers have received prizes at AFIPS and ACM conferences. An early leader in language standardization activities, he remains an active member of the IFIP Working Groups 2.3 on Programming Tools, 2.4 on Machine-Oriented Higher Level Languages, and 5.3 on Discrete Manufacturing.

John B. Goodenough is Director of Programming Technology at SofTech. He has been the principal technical participant in two language design studies and a research effort whose objective was to define current problems and the state of the art in language processing technology and modularity techniques. His current work focuses on the area of software reliability and software testing techniques.

Dr. Goodenough's experience includes 4 years with an Air Force psychology laboratory as a research mathematician and system programmer, and 5½ years with the Air Force Command and Management Systems at its Electronic Systems Division as a research mathematician. Here he was in charge of exploratory development programs in computer science, with emphasis on programming languages and automated program development techniques.

Dr. Goodenough received his PhD in Applied Mathematics (Computer Science) from Harvard University with a dissertation dealing with analysis and comparison of programming language syntax and semantics.

C. A. Irvine is a specialist in operating system design, software engineering, implementation languages, firmware architecture, distributed processing and computer networks, and systems measurement and analysis. As Director of Support Software Development at SofTech, he is responsible for developmental efforts for compilers, cross compilers, and integrated support software facilities, and has participated in the development of Structured Analysis.

Previously Irvine was with The National Cash Register Company's Data Processing Division as Manager of Software Architecture. He was a founder and formerly Senior Staff Scientist of Jacobi Systems Corporation, and has held technical and management posts with System Development Corporation, UCLA, Hughes Aircraft Company, and Space Technology Labs, Inc (now TRW Systems).

Irvine served for several months as head of the Century II Design Team which was responsible for the hardware/firmware/software design of a new product line family. As Chief Designer, he also directed the software designers charged with producing a design and implementation plan for a family of fourth-generation operating systems and support software.

SOFTWARE RELIABILITY AND DESIGN: A SURVEY

Peter Freeman
University of California, Irvine

60

Summary

This paper begins with an examination of the nature of software reliability, noting that two facets of reliability (correctness and robustness) are often confused. A brief survey is given of current attempts to improve program correctness and develop mechanisms that provide robustness. Starting from a discussion of the distinction that should be (but often isn't) drawn between design and programming, we identify six trends in software design that can be expected to impact significantly our ability to achieve reliable software. The paper closes with a brief discussion of the impact of software design automation on reliability.

Introduction

The ever-increasing expectations and needs of large organizations and the advent of large, cheap memories has led to the creation of ever-larger information systems. One of the results has been the discovery that while a small system could often be thoroughly tested for all practical purposes, large systems of interacting hardware, software and people could be rendered useless because of unreliability. Since the physical and economic consequences of information system failure may be very great, interest in reliability has grown also.

As with systems of any kind, we must be concerned with both the components of the system and their interactions. Hardware designers have long been concerned with reliability and, indeed, are able to create highly reliable hardware. The state of the art in software is much less advanced and we are only beginning to understand the issues involved.

Since software is the dominant component from a system control standpoint, we must not only understand the interaction between unreliable hardware and software systems, but we have the opportunity of improving system reliability by properly structuring the software. These issues are even less well understood.

Most people intuitively view as reliable, software that does not fail to do what they want it to do. When we consider large collections of programs, perhaps each with small errors, and other system elements which can introduce errors over which the software has no control, our intuitive view must be enlarged to consider software that has some probability of failing. Current software reliability research has two major thrusts: 1) bolstering our intuitive views with explicit (and sometimes formal) definitions, and 2) providing mechanisms and methods of achieiving relisble software.

Because software does not fail in the same way that a piece of hardware malfunctions, it is important to begin with a clear understanding of what is meant by "software reliability." Our survey then will concentrate on design because of the critical role it plays in bringing together the various aspects of software reliability.

While it contains information of interest to a wider audience, this paper is primarily a survey for professionals familiar with design and reliability in other areas who wish to know more about software design and reliability. Space prohibits our presenting a complete survey of techniques and the changing nature of the field makes it difficult to choose a particular example on which to concentrate. Instead, we will provide a structural overview that will permit the reader to understand some basic concepts and to then place in perspective other work on software design and reliability.

We first expliciate the meaning of reliability and mention several attempts to improve a system's reliability. Then we describe the stages of software creation and illustrate the importance of design to reliability. About half of the paper is then devoted to explaining several current trends in software design and discussing their impact on reliability. Possibilities for automating some of the techniques described are mentioned and several likely future developments are discussed.

The Nature of Software Reliability

There are many different and often confusing notions of software reliability in the literature. A quick survey of the papers from an important recent conference [ICRS, 1975], for example, will turn up many different definitions. The confusion stems from several things: concern with software reliability is recent, a naive view (often taken by programmers) focuses just on a program and not on the larger context in which it must operate, and software reliability does indeed consist of several interacting complex issues. Careful

consideration of what others have written a reliability and of the problems that m be solved, leads us to the following.

There are two basic concepts that make up what we loosely call reliability:

Correctness - A program is correct if it performs, properly the functions that we intended and has no unwanted side effects.

Robustness - A program is robust if it will continue to do something reasonable in the presence of environmental changes (such as hardware failure) and demands (such as bad data) that were not foreseen. In addition to robustness, the terms fault-tolerant and error-resistant are often used to describe this property.

Correctness is a more narrow concept since it refers only to the operation of a system with respect to conditions that we can lay down in advance. Since large systems are effectively infinite in terms of their possible inputs and states, it is clear that we cannot determine the correctness of a complex system for all possible situations. It is at this point that the wider concept of robustness enters.

Software reliability as we understand then, connotes a piece of software that is correct with respect to stated requirements and that, further, is able to withstand unanticipated demands as well. We will try to use the term reliability for the combined properties of being correct and robust, while reserving those terms for their more specific meannings.

Some have confused correctness with reliability, stating that the only problem is to ensure that programs work correctly. In fact, an incorrect program may still be considered reliable if it is robust and prevents errors from causing a disaster. For example, a payroll program might incorrectly calculate a little-used deduction while having careful checks on input data that prevent a bad input from causing system failure.

Correctness

Correctness is a property like beauty that is often mostly in the eye of the beholder. Software almost never turns out to be totally what the customer had in mind -- that is why formal specifications and contracts are needed. A customer may specify that a system with certain functional properties should be built. Because the environment in which it will be used is complex and so well known to the customer, he fails to spell out many of the things that he takes for granted. The resulting system may then turn out incorrect to him.

For example, the manager of an order-entry operation accustomed to taking care of a buyer's need, even if poorly stated, may fail to specify what actions are to be taken by a computerized system for direct order entry in the event of incorrect input. The system designer, not attuned to dealing with customers, may build a system that works well as long as standard catalog numbers are used but that provides a somewhat rude response if out-of-date numbers are used. The result: a system that seems correct to the builder, but woefully incorrect to the customer.

It is because of this possibility of subjective opinions on the correctness of a system that more precise notions of correctness are being developed. We will discuss these approaches below as one of the contributing factors to software reliability.

Robustness

Robustness is concerned with making programs well-behaved in the face of hardware failure, bad inputs, unexpected demands -- even incorrect operation of parts of itself. All software operates in an environment over which the builder of the software has little or no control -- hardware fails, other software interfaced ato the system works incorrectly, users provide bad input and overload the system. If we are concerned with reliability, we must make our software robust so that it can cope with such situations.

Coping may mean finding alternative ways of carrying out required functions even though something is wrong. It may mean notifying a higher authority that something is wrong. It almost always means not propagating the error so that problems are contained and catastrophe does not ensue. It may mean finding some way to recover from the malfunction.

It is important to stress that correctness alone is not sufficient. A perfectly correct program that does not check inputs to make sure they are within range may proceed to overwrite valuable files, producing highly unreliable behavior. Certainly a program must be largely correct before we call it reliable, but this is only a necessary condition, not a sufficient one. Robustness is an essential ingredient of reliability.

One catch is the term "intended," since we can intend for the software to operate properly at all times. This isn't cricket, however, and as noted below one of the current trends is to make operational our intentions for a program so that its correctness can be judged accurately.

Reliability Theory

There is a well-developed theory of reliability in other engineering areas [Lloyd and Lipow, 1962] which has been used extensively in the design of computer hardware [Shooman, 1968]. Several researchers [Trivedi and Shooman, 1975; Rorands, 1975] are attempting to apply some of these same reliability measures and estimation techniques to software. It should be clear from our discussion above that where possible our understanding of reliability would be greatly refined if we had precise measures of a system's reliability. The value of being able to estimate a system's future reliability on the basis of past performance is clear.

Like most of the work on software reliability, these formal studies are recent enough that we cannot accurately assess their eventual strength. The fact that software components do not have a failure rate that can be related to the underlying technology in the same way that the failure rate of hardware components can be means that new approaches must be developed. Taken in the aggregate, however, one can study overall failure rates and these may be related to software structure in some cases.

These formal approaches to studying reliability will certainly contribute to our ability of creating reliable systems and must be studied by one seriously concerned with software reliability. However, just as formal approaches to insuring the correctness of a program will never be able to provide us with totally reliable software, formal definitions and estimations of reliability will never be able to characterize completely what we intuitively understand.

Improving Reliability

In this section we will indicate the range of current efforts aimed at improving software reliability.

Correctness

Constructive programming is a term applied to any programming method that attempts to produce correct programs without the usual testing and debugging phases. Structured programming, top-down programming [e.g. Cowen and Kelly, 1975], and step-wise refinement [Wirth, 1973] are all constructive approaches that in many instances result in programs that are substantially more correct than ones produced in less organized ways. A fundamental objective of most work in programming methodology is to make programming more constructive.

Testing is the filter that is intended to determine if a program is correct, but often doesn't. Thus, any technique improving the effectiveness of testing will result in more correct programs. Automatic test case generators [Howden 1975], performance monitors [Stucki, 1973] and automated testing systems [Brown, 1975] all help us improve testing. Specialized testing procedures for particular classes of software can also be developed; several are described in [Hetzel, 1973]. (Hetzel also provides a good classification of testing techniques.)

As with constructive programming, many testing techniques and tools are in the development stage. Thus, it is still essential in most shops to make sure that programmers use standard manual techniques for testing [van Tassel, 1974; Trispnell, 1969].

Program verification techniques [London, 1975] are based on mathematical proof procedures and offer the advantage of permitting us to prove the correctness of a program without resorting to testing. Strictly speaking, program verification is a process of proving mathematically whether or not a program will fulfill a set of assertions about its operation when it terminates (which is treated as a separate problem). The assertions are assured to express our requirements for correct operation of the program so that by proving the program obeys the assertions, we will have proved its correctness.

Unfortunately, there are two problems with this approach which have not been entirely overcome: First, we may make a mistake in stating the assertions so that even though the program is shown to meet them, we may have failed to capture our intuitive understanding of correct operation for the program. Second, the proofs themselves may be quite complicated and thus are open to error if done by hand or are beyond the reach of current automatic methods.

The underlying reliance on mathematical reasoning, however, is closely tied to the original notions of constructive programming as put forth by Dijkstra (see [Freeman, 1975, p. 487] for more discussion). One can develop the assertions before actually programming and then write the programs to make the assertions true. A large amount of current research is aimed toward developing proof techniques (both formal and informal) and toward developing practical programming languages that facilitate such proofs. This research is too extensive to review here, but can be expected to provide significantly improved methods of obtaining programs that are correct (in the broader sense of meeting our expectations, not only our formally stated requirements).

A middle ground between completely formal program proof techniques and completely manual approaches uses formal techniques related to practical and human-aided tools [Good, et al, 1975; Stucki, 1975]. This approach may well provide us with the means to produce highly correct programs. Most automated program verification

niques are still a long way from being ...ily practical and most manual or semi-automated techniques are not widely used because they still require a good deal of effort and care to use properly.

This brief review of techniques for constructing correct programs may give you the impression that no one has thought about the problem until recently and that it is all still black magic. While programmers have always tried to build correct programs, it is true that explicit concern with program construction techniques and tools for aiding the process are recent and that for the most part we are only beginning to utilize improved methods.

Robustness

Improving our ability to build correct programs is a very general task. Improving our ability to build robust programs provides a much more focused goal: We must devise mechanisms that will permit software to cope with whatever unexpected occurrences threaten its operation. Some of the reliability mechanisms are really no more than just good programming practices that have been used for years: checksums, checking of parameter ranges, data validity checks, and so on. (Goodenough's [1975] thorough consideration of exception handling is relevant in this regard). Wulf [1975] describes several mechanisms used in the Hydra operating system to provide robustness, but feels compelled to apologize for mentioning them since they all seem so obvious. Yet, the sad fact is that many programmers do not use such mechanisms and there are really no good "handbooks" one can consult for techniques. Thus, until an encylopedic treatment of reliability appears, you must laboriously gather good mechanisms from the descriptions provided by others or else invent your own.

Let us mention just a few here to illustrate the type of structures that provide robustness. Self-identifying information structures provide protection against some types of hardware failure or the accidental destruction of pointers to the structure. The simplest example is a disk file that includes a header containing the name and other identifying information of the file; if the directory that points to the file is destroyed, it can be recreated from the information in the file itself.

Modularity (see Freeman [1975] p. 490 for discussion) provides an opportunity for improving robustness. As data and parameters are passed between modules, the opportunity exists to check them against expected values and thus detect problems before they propagate. This and the following mechanism provide what are often called firewalls.

Many of the mechanisms developed for making systems secure, such as capabilities [Dennis and Van Horn, 1966; Fabry, 1974], also provide increased reliability. When resources are carefully controlled, for example, checks of resource ownership will be performed. This provides an opportunity to catch erroneous (not just illegal) system operation through the detection of internal errors.

A new structure developed specifically to provide robustness is the idea of recovery blocks [Randell, 1975]. Simply, this is a means of indicating portions of a system whose operation must pass a dynamic acceptance test designed to determine proper operation; if the test fails, alternate means of achieving the desired result are indicated and automatically tried. (This mechanism obviously cannot be applied everywhere since it assumes a computation can be retried).

As structures providing robustness are developed, the task of making systems reliable will become correspondingly easier. The critical role of design in providing overall reliability will remain, however, and it is this factor that we now investigate.

The Software Creation Process

Although there are many characterizations of the system design process (see, for example, [Teichroew, 1974b] and [Yourdon, 1975]) we believe these five main stages in the life cycle of a piece of software provide a good basis for discussion:

1) Needs analysis - The eventual user of the software discovers a need for some software; the nature of the need is analyzed; the outlines of the type of software that would satisfy these needs are established. This may be a formal development stage or it may be carried out at the same time specifications are produced.

2) Specification - Functional descriptions of the software are developed, along with constraints on its structure and resource usage. Economic constraints on the development process itself are stated. This stage may be combined with the needs analysis and in many instances is blended into the design stage in an iterative sequence that goes back and forth between statement of specifications and refininement of the design.

3) Design - A representation of the system is developed. This representation often exists at several levels of detail typically called overall or system design, detailed design, module design, and so on. Major parts of the system are determined and named, logical relationships of control and data between parts are established, and the operation of individual pieces is determined. This representation may be carried to a level of detail approximating programs, but stopping short of making all decisions normally associated with programming (housekeeping details, local data structures*

---

*See Chapter 13 of [Freeman, 1975] for a detailed discussion.

precise flags, keys, and so on.)

4) Implementation - Programming, testing individual pieces; integration of pieces to form sub-assemblies, checkout of those pieces; system testing and performance measurement. Debugging, redesign, and changes to make the actual system conform to specifications.

5) Maintenance - Everything after the system is "finished": location and repair of bugs, addition of functions, alteration of existing functions.

It is important to point out that one almost never goes through these five stages without backing up. The boundaries between them are often blurred and it is usually necessary to iterate between several levels. a new type of system, it may be necessary to attempt an implementation in order to get information critical to making design decisions. Further, most large systems will have several of the activities underway simultaneously for different parts of the total system.

The precise delineation and naming of activities in the cycle is unimportant for our purposes here. Most people have their own structuring of the cycle which looks basically like that presented above but which may vary slightly or use different names indigenous to their own environments. Before considering trends in software design -- really specification and design as outlined above -- and their impact on software reliability, we want to discuss briefly the nature of software design and how it differs from programming.

Design vs. Programming

We want to characterize the activities in software design in a general way to emphasize that we are not just talking about programming. First, consider the several activities involved in programming at the lowest level (coding):

-devising local and concrete data representations for information;

-forming precise algorithms for doing necessary processing;

-taking care of housekeeping details necessitated by the particular programming system used (language plus run-time environment);

-choosing names, forming syntactically correct language statements, and making the program letter perfect.

By contrast, design, especially at the higher levels, is concerned with somewhat different activities:

-abstracting the operations and data of the task situation so that they may be represented in the system;

-determining precisely what is to be done by the software under design;

-establishing an overall structure of the system;

-establishing interfaces and definite control and data linkages between parts of the system and between the system and other systems;

-choosing between major design alternatives;

-making tradeoffs dictated by global constraints and conditions in order to meet varied requirements such as reliability, generality, user-centeredness, and so on.

There is an easy way of distinguishing the two activities. When programming, we are constructing programs (often in some higher-level language). We must bind data representations and control sequences so that the program will execute properly. When designing, we are devising representations of programs, not actual programs. Clearly we will be doing some of the same things in both activities -- we may choose some definite control sequences during design, for example -- and detailed design often comes arbitrarily close to producing complete programs (especially when using a metacode - see below).

Another way of emphasizing the difference between programming and design is to consider what is actually produced -- that is, what is on the pieces of paper produced by a programmer or a designer. A programmer produces programs and documentation of those programs. The documentation is needed to make the programs understandable to humans but it is not a necessary condition of existence. That is, the programs exist without the documentation, they may execute without the documentation, and indeed we instruct someone studying a program to "go to the code" for the final word on what the program contains.

By contrast, when designing we produce representations of programs. These representations may be very high-level in detail (e.g. an overall black-box diagram representing major control-flow) or very near to actual programs (e.g. a metacode description of a program that is complete except for minor housekeeping details).

---

* In [Freeman, 1976] we discuss some of the technical issues involved in using such representations.

That design is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher levels and on the logical operations involved at the lower levels. Programming is then the process of turning such a representation into a program that will execute on a specific machine.

While the distinction between design and programming seems obvious, it often hasn't been to those building systems. In our following consideration of trends, bear in mind that we are most concerned with design and its impact on reliability.

Trends in Software Design

Software design techniques are undergoing active investigation and change. While these changes are only beginning to affect significant numbers of people, several important trends that will directly affect reliability can be identified.

More Design -- Less Coding

The most profound and widespread trend is the increasing concern with the specification and design process. People are realizing that the difficulties experienced in creating large software systems are not due solely to bad programming practice or insufficient management.

It is true that it is often possible to create a small program without engaging in much formal design activity. A craftsman programmer can usually create an acceptable small program without much supervision and without engaging in much documentation of alternatives (i.e., design).

We might compare this to craft production of artifacts in other areas. For example, a boat builder may create a small boat similar to ones produced before, but with some variation, without drawing detailed blueprints. But a shipbuilder cannot build an aircraft carrier successfully without engaging in a large amount of design -- consideration of alternatives, careful fitting of form to constraints, checking that all that is required has been included, and planning of the actual implementation.

We now realize that the same is true of building large software systems (and sometimes even small ones of high complexity). The tasks we have set for ourselves demand design skills and techniques that go beyond the valuable new programming methodologies, such as structured programming.

One result has been increased awareness of the important role that design must have. Studies of where errors occur [Thayer, 1975] and where the actual effort on a project goes [Boehm, 1973] have also pointed us in the direction of design. As people have become concerned with properties of software, such as reliability, then it has become even clearer that these properties must be designed into a system from the start and that trying to add them on at the programming stage will not work.

The primary impact on software reliability that this increased emphasis on design will have is to provide an opportunity in the software creation cycle for proper consideration of reliability. Although reliability is our interest here, it should be clear that creating a software system involves many factors: (e.g., generality, portability, and maintainability) which may deserve careful consideration; acceptance of the software by the eventual user must be considered and often goes beyond technical qualities in importance; and economic factors typically cover all other considerations in the sense that for any given property (such as reliability), the amount of it we get is usually a direct function of how much we are willing to pay.

When software is slapped together with most of the time spent on testing and retrofiting brought about by precipitious coding, there is little opportunity for considering the inevitable tradeoffs between various qualities and for making sure that features that must be dealt with on a system-wide basis (e.g., reliability) are considered. The hallmark of design is the consideration of alternatives and the weighing of conflicting demands to find acceptable compromises. As we move toward more extensive design, we at least will have the opportunity to consider reliability and to do it on a system-wide basis.

Coherent Methodologies

The current interest in software design focuses on two areas: developing coherent methodologies for design and developing tools to aid or augment the designer. We will briefly address this second focus in a later section. A methodology is a collection of methods (techniques, procedures) to follow, which if faithfully carried out and applied in a particular situation will (in theory) achieve some goal (e.g., the attainment of a correct design). A methodology is more than a recipe. It usually consists of several aspects, may not be complete, and it typically cannot be applied blindly.

For example, a design methodology might prescribe the order in which certain classes of decisions are to be made, ways of making decisions, ways to represent the developing design, and so on, but not address project management. The assertion is then that if the prescriptions are followed one will be able to achieve a correct design more easily.

Several new and varied design approaches have been put forward in recent years. The chief-programmer team concept of IBM [Baker, 1972], while concentrating on programming, clearly involves design activity as well. In particular, the chief programmer can be viewed more as a designer (in the terms we have been using) than as a programmer. TRW has reported substantial success with their Size Defense methodology [Williams, 1975] developed on a large military project. The idea of structured design [Stevens, et al, 1974] has gained a number of adherents. ISDOS [Teichroew, 1976], developed by Teichroew and his colleagues, also is gaining popularity as a system design technique. Lucas [1974] has cogently proposed and put into practice the creative systems design methodology that takes into account the impact that a computer system has on an organization. Rice, et al [1974] have developed and used a total methodology that touches all phases of the creation cycle.

These and other methodologies share several characteristics of a general nature:

- they structure the decision making process so that similar decisions can be made together;

- they provide various means of making the design and its progress more visible;

- they prescribe management and/or organizational procedures to be used in large design projects;

- they stress the need for a clear statement of objectives as well as a clear statement of the design itself.

- they match the activities of different phases of the creation cycle to form a coherent and harmonious whole.

None of these methodologies are a panacea and none are yet fully developed. However, all of them, and others as well, typically offer some improvement over completely ad hoc methods.

The impact of a coherent methodology should be clear: Producing a reliable system is an activity that must be spread over the complete creation process. When that process is composed of mismatched techniques and when the overall management controls permit some phases to be missed then producing reliable software will be next to impossible. Coherent methodologies eventually offer solutions to these organizational causes of unreliable software.

Modularization

Breaking a program or system into pieces, or modules, is one of the oldest concepts in computing. Yet, it is also one of the current trends in software design. The definition of what constitutes a module is now seen as an important design parameter. Mills [1971] and others argue strongly that segments of code should be severely limited in size to facilitate understanding. Parnas [1972b] has suggested that modules be used to hide design decisions and that we must modularize a system in ways which make later changes in the design easy. It is clear that modularization is a necessary condition for pragmatic program verification.

Whereas, previously modularization was often simply used to break a task into pieces small enough for a single person to work on or small enough to fit into a memory space, we now see that modularization will have a very real effect on our ability to understand and deal with the complexity in a system. Modularization is just the implementation of the "divide and conquer" approach to problem solving and is essential to techniques like chief-programmer teams. Thus, it is not surprising that if we do not divide the system appropriately then it may be more, rather than less, difficult to solve the design problems involved.

Modularization has a very direct impact on program correctness by reducing complexity into manageable units and thus reducing the chances of error. Further, modularity permits application of formal verification techniques to programs small enough that the proofs are manageable.

The impact on robustness can be gained through the use of components. Small, highly robust modules can be developed and then reused in other systems. Since they are known to be reliable, the task of producing a new reliable system is reduced.

Formal Specifications

A fourth trend in software design, not yet so well developed but receiving a large amount of attention, is the use of formal specifications. Earlier we noted the difficulty of assessing the correctness of software in many instances because it is not clear what the software is supposed to do. In response to this some have attempted to develop more explicit ways of specifying the goals for a piece of software.

The Problem Statement Language (PSL) used in ISDOS [Teichroew, 1974a] is a language for stating software requirements in an unambiguous and functional manner. More formal approaches, typically using mathematical logic to state requirements, have been tried but have not yet been turned into practical tools. The assertion languages (again, often based on predicate calculus) used to state correctness assertions for use in program verification are a way of formally specifying what a program is to do.

Parnas has developed a technique for specifying modules [1972a] and this technique has been coupled with a comprehensive design

methodology by a group at SRI [Robinson, 19 to attack a significant sized problem (t design of a secure operating system). Rob...son [1976] addresses this entire issue in more detail. The brevity of our comments on this issue are due to the fact that his paper is being presented at the same session as this paper and should not be construed as an indication that specification is unimportant -- quite the contrary!

The primary impact of formal specification will be on correctness. Not only will such techniques, when further developed, make it easier to check formally the correctness of a system, but the existence of clear and unambiguous specifications will help focus the efforts of the designers, resulting in designs of better quality.

Design Verification

A trend only beginning is that of design verification. The idea is simple and in fact has been practiced for many years: We would like to test that a design is correct before we turn it into the actual object. Design reviews are intended to do this. But, they don't fully succeed because the complexity of most designs and our inability to predict the implications of software design decisions makes it impossible to verify designs with any accuracy using informal means.

Techniques that permit us to verify the correctness of a design with some accuracy will have a great impact on reliability. Not only will they help us to improve vastly the correctness of our software, but they may help us see the results of various possibilities before building the system. This will permit us to build in mechanisms to provide robustness on the basis of this early feedback concerning possible vulnerabilities.

Metacode representation

We noted above the importance of representation in software design since in designing, we are solely building a representation, not the final object. A very valuable trend in software design is to use a programming language-like notation in which to express a software design, especially at the levels of most detail. These notations are usually called a metacode or program description language (PDL) [Caine and Gordon, 1975; Freeman, 1976].

There is no standard metacode at present (indeed, there may never be) but most share some common features:

-standard control structures, usually limited to those considered well-structured (do-while, if-then-else, case, etc.)

-block structure;

-abstract data types;

-local and global data structures, parameters;

-conditionals and operational statements expressed in English (perhaps a restricted subset);

-regular programming language statements as needed.

Most metacodes are based on Algol-68 or PL/1. They use the textual structure and control mechanisms of the base language while permitting more freedom in the expression of conditions and operations. This freedom may range from completely unrestrained use of English to highly constrained use of predefined and mnemonically named functons. Figures 1 to 3 provide a brief example.

```
procedure filecommands(input-command)
   do while more input
      if command not legal then provide error
         message;
      else
         do case (input-command; 'OPEN':open,
            'PROTECT',protect,...)
```

Figure 1: High-level Metacode Representation of a Command Decoder

```
procedure protect(file,owner,user)
   begin
      check for parameter validity;
      if not valid then exit with error code;
      mark file read-only;
   end
```

Figure 2: High-level Metacode Representation of 'Protect' Command

```
procedure protect (file,owner,user)
   begin
      if owner = user then exitwith('user =
         owner');
      if file is open then exitwith('file in
         use');
      get directory for owner;
      if none then exitwith('owner does not
         exist');
      find file entry;
      if none then setreturncode('file does
         not exist');
      else begin
         mark file read-only;
         setreturncode('done');
         end
      release directory
   end
```

Figure 3: Detailed Metacode for 'Protect' Command

Metacode makes visible many of the logical decisions concerning control and data before the lowest level programming details are added. This permits the designer to check the design for logical correctness more

easily since the metacode is usually much easier to comprehend than a full [illegible] representation. This aids in verification not only to achieve correctness, but also to assess the reliability of the system.

Because the metacode has the same structure as a programming language, most people find it extremely easy to code from it. Basically, the use of metacode has the effect of pushing detailed design and debugging up a level into a representation that facilitates, rather than hinders, obtaining correct and robust programs.

Automation of Software Development

In a paper several years ago [Freeman, 1973], we pointed out several ways in which software design could be augmented or automated. Those general approaches, if realized, would help us enhance the reliability of systems in ways similar to those described above.

While the primary purpose of this paper is not to discuss design automation, the nature of the meeting for which it was prepared makes it appropriate to consider briefly ways in which software design can be automated or augmented. Further, most would agree that achieving reliability of both types in complex systems will require augmentation of human capabilities.

Several areas of software design with special relevance to achieving reliability that could benefit greatly from automated aids are:

Processing of representation: The design of large systems involves a large amount of information. Designers should be relieved of burdensome clerical duties and their activities enhanced by simple aids such as cross-indexing and syntax checking of designs. This will permit them to focus more on the content of the design.

Checking of consistency and completeness: Usage of the same information in different places in a design should be consistent. Routines called as subroutines with particular parameters should be checked to make sure they are set up to handle those parameters. Functions or data structures used in one part of a design must be specified elsewhere. These and other questions of completeness and consistency can be checked automatically in many instances.

Comparison to constraints: Design is a process of creating an artifact within constraints. If those constraints are formally stated then a developing design, properly represented, could be checked automatically for conformity. In large systems with many, detailed constraints, such automated checking is essential.

Comparison to specifications: Again, as we develop formal means of specifying the functions of a system, techniques can be developed for automatically checking a proposed design against the specifications it is supposed to meet.

Suggestion of alternatives and ramifications: Another fundamental characteristic of design is the choice between alternatives. If a design is developed in a machinable representation and if one has built up a library of possible structures, then a design-aid system could suggest alternatives to the designer. This would be especially applicable to providing suggestions with known reliability properties. Additionally, when design decisions are made, an automated monitor could determine if anything is known about the characteristics of this structure (e.g. its susceptibility to error) and suggest ramifications of the decision to the designer.

These brief descriptions of areas in which automation and/or augmentation look reasonable should suffice to indicate ways in which trends in software design may impact our ability to build reliable systems. More complete treatments can be found in [Ramamoorthy and Bo, 1975] and [Reifer, 1975].

Conclusion

Other trends can be discerned in software design as well: more use of prototypes, use of prepackaged componenets, simulation, and formal decision techniques. Some of these may have as much or more impact on design and reliability than the trends we have chosen to survey here.

In addition, there are other active areas of research that will undoubtedly affect our ability to produce reliable software: , programming language design, operating system structures and design techniques, and abstract data types. The latter area, especially, will have a large impact on the way people approach software design and thus on the quality of the systems they produce.

Three things seem clear for the future: 1) additional mechanisms will be developed that provide robustness; 2) the trends in design mentioned above will continue to grow in importance; 3) augmentation by computer of software design will grow in importance, especially in the design of very large systems.

Acknowledgement

Many helpful comments were received on a draft of this paper. I am deeply grateful to several colleagues for their suggestions, especially Sue Gerhart, Beverly Johnson, Steve Levin, Hank Lucas, Ralph London, Tim Standish, Leon Stucki, Dan Teichroew, Tony Wasserman, and Ray Yeh. Shirley Rasmussen did an admirable job of typing this paper on a computer-based editing system.

References

Baker, F. T. 1972. "Chief Programmer Team Management of Production Programming." IBM Systems Journal. 11,1.

Boehm, Barry W. 1973. "Software Design and Structuring." In [Borowitz, 1975].

Brown, J. R. 1975. "Getting Better Software Cheaper and Quicker." In [Borowitz, 1975].

Buxton, John, Peter Naur, and Brian Randell (eds.) 1969. Software Engineering Concepts and Techniques. Petrocelli/Charter (reprinted in 1976).

Caine, Stephen H. and E. Kent Gordon. 1975 "PDL - A Tool for Software Design." Proc AFIPS NCC 1975, p. 271.

Dennis, J. B. and E. C. Van Horn, 1966. "Programming Semantics for Multiprogrammed Computations," CACM 9,3 pp. 143-145.

Fabry, R.S. 1974. "Capability-Based Addressing," CACM 17,7 p.403.

Freeman, Peter. 1973. "Automating Software Design." 9th Design Automation Conference, Reprinted in Computer, April, 1974.

Freeman, Peter. 1975. Software Systems Principles, Science Research Associates.

Freeman, Peter. 1976. "Software Design Representation: Analysis and Improvements." Submitted for publication and available from the author.

Good, D. I., R. L. London, and W. W. Bledsoe. 1975. "An Interactive Program Verification System." In ICRS [1975] p. 482.

Goodenough, John B. 1975. "Exception Handling: Issues and a Proposed Notation," CACM 18,12 p. 683.

Hetzel, W. C. (ed) Program Test Methods. Prentice-Hall.

Hice, G.F., W.S. Turner, and L.F. Cashwell. 1974. System Development Methodology, North-Holland/American Elsevier.

Borowitz, Ellis (ed.) 1975. Practical Strategies for Developing Large Software Systems. Addison-Wesley.

Howden, W. E. 1975. "Methodology for the Generation of Program Test Data". IEEE Trans. Computers, May.

ICRS. 1975. Proc. International Conference on Reliable Software. Appeared as SIGPLAN Notices 10,6 (June).

Lloyd, D.K. and M. Lipow. 1962. Reliability: Management, Methods, and Mathematics. Prentice-Hall.

London, Ralph. 1975. "A View of Program Verification." In ICRS [1975], p.534.

Lucas, H. 1974. Toward Creative Systems Design. Columbia University Press.

Mc Gowan, Clement and John Kelly. 1975. Top-Down Structured Programming Techniques. Petrocelli/Charter.

Mills, H. D. 1971. "Top-Down Programming in Large Systems." In Debugging-Techniques in Large Systems. R. Rustin (ed.) Prentice-Hall.

Moranda, P. 1975. "Predictions of Software Reliability During Debugging," Proc. Reliability and Maintainability Symposium, Washington, DC.

Parnas, D. L. 1972a. "A Technique for Software Module Specifications with Examples." CACM 15,5

Parnas, D. L. 1972b. "On the Criteria to be Used in Decomposing Systems into Modules." CACM 15,12.

Ramamoorthy, C. V. and S. F. Ho. 1975. "Testing Large Software with Automated Software Evaluation Systems." In ICRS [1975] p. 382.

Randell, B. 1975. "System Structure for Software Fault Tolerance." In [ACM 1975] p. 437.

Reifer, D. J. 1975. "Automated Aids for Reliable Software." In ICRS [1975] p. 131.

Robinson, L., K. Levitt, P. Neumann, A. Saxena. "On Attaining Reliable Software for a Secure Operating System", In ICRS [1975], p. 267.

Robinson, L. 1976. "Specification Techniques for Software Reliability". Proc 13th Design Automation Conference.

Shooman, M.L. 1968. Probabilistic Reliability: An Engineering Approach, Prentice-Hall.

Stevens, W. P., G. J. Myers, and L. L. Constantine. 1974. "Structured Design," IBM Systems Journal.13,2.

Stucki, L. G. 1973. "Automatic Generation of Self-Metric Software." 1973 IEEE Symposium on Computer Software Reliability.

Stucki, L.G. 1975. "New Assertion Concepts for Self-Metric Software Validation." In ICRS [1975], p. 59.

Teichroew, D. 1974a. "Problem Statement Analysis: Requirements for the Problem Statement Analyzer (PSA)" in J. D. Couger and R. W. Knapp, System Analysis Techniques. John Wiley and Sons.

Teichroew, D. 1974b. "Improvements in the System Life Cycle." Proc. 1974 IFIPS Congress, pp. 972-978.

Teichroew, D. 1976. "ISDOS and Recent Extensions." Proc. 1976 Brooklyn Polytechnic MRI Symposium.

Thayer, T. A. 1975. "Understanding Software Through Analysis of Empirical Data." Proc. AFIPS 1975 NCC.

Trapnell, F. M. 1969. "A Systematic Approach to the Development of System Programs." Proc. AFIPS 1969 SJCC, p. 411.

Trivedi, A.K. and M.L. Shooman. 1975. "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters." In ICRS [1975], p. 288.

van Tassel, D. 1974. Program Style, Design, Efficiency, Debugging, and Testing. Prentice-Hall.

Williams, R.D. 1975. "Managing the Development of Reliable Software." In ICRS [1975], pp. 3-8.

Wirth, N. 1972. Systematic Programming: An Introduction. Prentice-Hall.

Wulf, W. A. 1975. "Reliable Hardware-Software Architecture." In ICRS [1975] p. 402.

Yourdon, Edward. 1975. Techniques of Program Structure and Design. Prentice-Hall.

An Extendable Approach to Computer-Aided Software Requirements Engineering

THOMAS E. BELL, MEMBER, IEEE, DAVID C. BIXLER, AND MARGARET E. DYER

Reprinted from *IEEE Transactions on Software Engineering*, January 1977, © IEEE 1977.

Abstract—The development of system requirements has been recognized as one of the major problems in the process of developing data processing system software. We have developed a computer-aided system for maintaining and analyzing such requirements. This system includes the Requirements Statement Language (RSL), a flow-oriented language for the expression of software requirements, and the Requirements Engineering and Validation System (REVS), a software package which includes a translator for RSL, a data base for maintaining the description of system requirements, and a collection of tools to analyze the information in the data base. The system emphasizes a balance between the use of the creativity of human thought processes and the rigor and thoroughness of computer analysis. To maintain this balance, two key design principles—extensibility and disciplined thinking—were followed throughout the system. Both the language and the software are easily user-extended, but adequate locks are placed on extensions, and limitations are imposed on use, so that discipline is augmented rather than decreased.

Index Terms—Automated simulation generation, automated tools, requirements language, REVS, RSL, simulation, software engineering, software requirements, software requirements engineering, SREM, SREP.

I. Introduction

THE development of data processing systems has too often resulted in cost overruns, schedule slippages, or failures to produce a system which satisfies the original requirements [1]. As Boehm stated, "Software (as opposed to computer hardware, displays, architecture, etc.) is 'the tall pole in the tent'—the major source of difficult future problems and operations; performance penalties" [2]. His Air Force study showed that the annual expenditures by the Air Force and NASA for software are twice the expenditures for hardware. This indicates the tremendous impact that problems in developing software can have on the budgets of the services or on the cost of any large software-based system that is being developed.

A number of studies have identified the symptoms and suspected root causes of problems in software development. Among these are the McGonagle [3] study for the Air Force, studies by Bartlett *et. al.* [4], studies on software problems by Thayer *et. al.* [5], [6], and a study by Bell and Thayer [7]. In addition, studies by The MITRE Corporation [8] and the Applied Physics Laboratory of The Johns Hopkins University [9] concentrated on how software is managed. Foremost among the problems in software development identified in these studies is the generally undisciplined approach which is usually taken.

The Ballistic Missile Defense Advanced Technology Center (BMDATC) is sponsoring an integrated software development research program [10] to improve the techniques for developing correct, reliable BMD software. Reflecting the critical importance of requirements in the development process, the Software Requirements Engineering Program has been undertaken as a part of this program by TRW Defense and Space Systems Group[1] to improve the quality of requirements specifications. The product of this program (the Software Requirements Engineering Methodology, SREM) includes techniques and procedures for requirements decomposition and for managing the requirements development process. In addition, SREM includes the Requirements Statement Language (RSL), a machine-processable language for stating requirements, and the Requirements Engineering and Validation System (REVS), an integrated set of tools to support the development of requirements in RSL. SREM was designed to bring the computer's aid to the requirements engineering phase of software development in order to reduce the number and severity of problems encountered there. The purpose of this paper is to describe the type of automated system SREM needed to be, the characteristics of SREM that we chose to meet these needs, and how we have checked our work to be sure that both human creativity and computer-imposed discipline are achieved.

II. SREM Needs for Automation

Developing software requirements is generally a difficult intellectual job, and the job begins to look nearly impossible when it involves large data processing systems like those in a Ballistic Missile Defense (BMD) system. The requirements document for the current BMD system (the System Technology Program, Site Defense Project) contains 8248 requirement and support paragraphs in a 2500-page specification. Manually checking each paragraph against all others is an enormous task, and generating them initially is even harder; automated techniques are clearly needed. However, these automated techniques should not force the engineer to spend large amounts of mental energy dealing with simulators and control languages; he should be able to state the requirements in a reasonable, natural language and then have the automated techniques keep track of the changes, ensure consistency, and report the interactive effects of his statements.

Manuscript received July 22, 1976; revised September 14, 1976.

T. E. Bell was with the TRW Defense and Space Systems Group, Redondo Beach, CA. He is now with the Management Consulting Department, Peat, Marwick, Mitchell & Copartners, New York, NY.

D. C. Bixler is with the TRW Defense and Space Systems Group, Redondo Beach, CA 90278.

M. E. Dyer is with the TRW Defense and Space Systems Group, Huntsville, AL 35805.

[1] Under Contract DASG60-75-C-0022.

In order to allow the requirements engineer to use his creativity, the system should be natural and flexible; it should allow to express requirements in terms of concepts which are [illegible] to him. If severe restrictions are built into the system, such as highly constrained syntax rules or a paucity of concepts, the engineer would spend more time trying to "work the system" and less time working the requirements development problem. In addition, the technologies involved in a BMD system are so extensive and advance so rapidly that no predefined set of concepts could ever be expected to satisfy the specific needs of all future projects. Therefore, such a system should be extensible at the concept level so that a particular project with an application requiring a new concept may add it to the system.

A computer-aided system should enforce some measure of discipline on the creativity of the engineer so that the development process always moves in the direction of reduced ambiguity and increased consistency. For example, the computer could perform static checking of the requirements to illuminate inconsistencies such as conflicting names, improper sequences of processing steps, and conflicting uses of items of information which must be present in the system. With a flow orientation, the computer could additionally check the dynamic consistency of the system through the use of a simulation.

The Software Requirements Engineering Program has produced a computer-aided system for the development of requirements which fulfills the needs described above. The following sections of this paper provide an overview of that system, followed by descriptions of the Requirements Statement Language, the Abstract System Semantic Model (the central repository for requirements), and the tools that provide automated aid for the engineer.

III. Overview

The Requirements Engineering and Validation System (REVS) consists of three major segments:

1) a translator for the Requirements Statement Language (RSL),

2) a centralized data base, the Abstract System Semantic Model (ASSM), and

3) a set of automated tools for processing the information in the ASSM.

A diagram of the system is shown in Fig. 1.

Central to REVS is the ASSM; a relational data base similar in concept to the system used in the ISDOS Problem Statement Language/Problem Statement Analyzer (PSL/PSA) system [11], [12]. However, our need for extensibility and configuration management, as well as the flow approach needed for simulation, have necessitated many differences from the concepts utilized in PSL/PSA, and therefore differences in data base design.

The design of the ASSM provides a decoupling between the input language, RSL, and the analysis tools. This decoupling permits extending RSL without having to consider issues such as controlling the tools, interfacing with the host operating system, and doing other things which would compromise the naturalness needed in RSL. The decoupling has also permitted us to exercise great freedom in the design of RSL; we were free to develop the most natural way of expressing requirements without making concessions to control language or problems of configuration management.

RSL is designed to be a means for stating requirements naturally while still being rigorous enough for machine interpretation. We pursued this goal, in part, by orienting the design around the specification of flow graphs of required processing steps. These flow graphs are expressed in RSL in terms of "structures," which are the products of a mapping of a two-dimensional graph (e.g., Fig. 2) onto a one-dimensional stream suitable for computer input (e.g., Fig. 3). Structures are built from primitive flow specification blocks much as the control flow of a computer program is built from control specification primitives. The types of structure primitives available in RSL are fixed in order to provide discipline by precluding the formation of structures whose meaning may be unclear. The specification of the processing steps themselves, and of other information related to these processing steps (e.g., the data items which they use), is done in a much more flexible manner. In fact, the concepts which may be expressed in this nonprocedural segment of RSL are not necessarily fixed. The language is extensible at the concept level in order to respond to situation-specific needs and new, unanticipated needs for stating requirements. The structures and the nonprocedural statements of RSL are input to REVS through a translator which analyzes them to ensure individual correctness. The meaning of the statements is then abstracted and entered into the ASSM; no executable code is generated, only entries in the data base that can be used by the tools.

The tools' designers are freed from the syntax of RSL; they work with the abstracted information in the ASSM. Thus, a syntax change in RSL does not usually compromise any of the tools; tools may be added or modified as the Software Requirements Engineering Methodology (SREM) evolves, or as the application of REVS changes. The tools merely access the ASSM and in no way are dependent on RSL syntax.

The information available in the ASSM will support a wide variety of analysis tools. We have implemented a baseline set of widely applicable tools which perform analyses primarily related to flow properties of the information in the specification. Our analysis of requirements problems [7] indicated that these capabilities are very important in a methodology like SREM [13] for generating consistent, correct requirements and enforcing the desired discipline on the requirements generation process. Among these tools are an interactive graphics package to aid in the specification of the flow paths, static consistency checkers which primarily check for consistency in the use of information throughout the system, and an automated simulator generator and execution package which aids in the study of dynamic interactions of the various requirements. Situation-specific reports and analyses which a particular user may need in order to augment the information given by the baseline tools are generated through the use of a generalized extraction and reporting system. This system is independent of the extensions to RSL so that new concepts added to the language may be included in queries to the data base.

A unifying concept throughout the SREM, including RSL and REVS, is the specification of requirements for software in

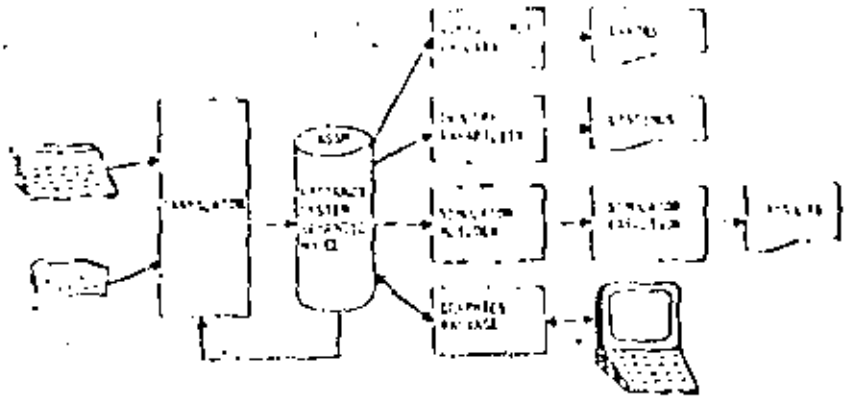


Fig. 1. Information flows in REVS.

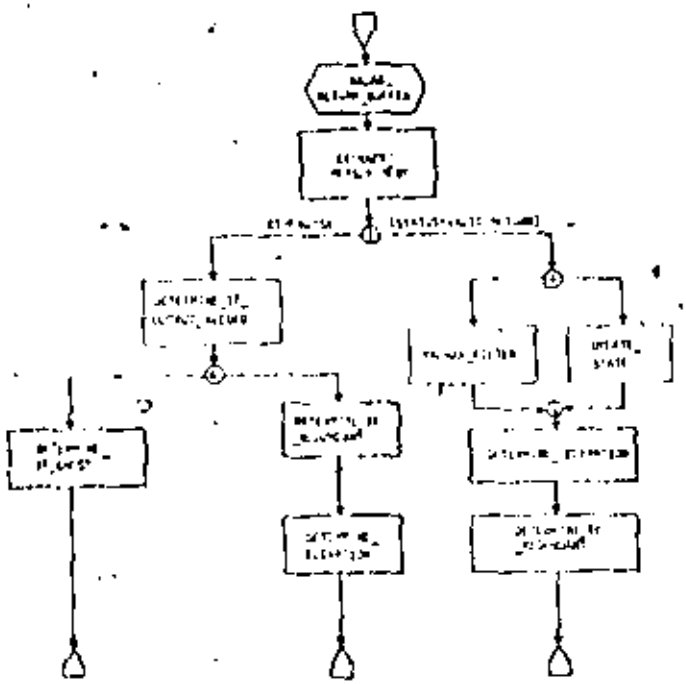


Fig. 2. Flow graph of a sample PROCESS.

of flows through the system [14]. The use of these flows helps to bring about a disciplined approach to the development of software requirements, much as structured programming has done for the implementation of software.

74

IV. RSL

Current software requirements documents are typically written in natural English and provide information about relatively isolated portions of the data processing system. Some of these pieces of information are very nebulous, and some are in such great detail that they represent an implementation—an implementation that may have been chosen without consideration of the remainder of the data processing system. The unevenness alone is annoying, and the probability is high that some parts of the system will be left without any written requirements. The largest deficiency of such documents, however, is that they often fail to provide critical information about how the pieces of the system will fit together. The implementor is often left without adequate information, and the requirements engineer cannot be certain the various pieces are consistent; often they are not, and no discipline is applied to make them consistent.

Our approach involves writing requirements in RSL [15], an artificial language. RSL provides information on how pieces of the system will fit together through use of the flow approach to defining the requirements. With this approach, the connectivity information is central to the development, and consistency may be easily checked, often through automated means. Another benefit of using an artificial language is the ability to define precisely the meaning of concepts in the language. With a natural language such as English, each person has a different idea about the connotation of a particular phrase and these differing ideas lead to ambiguities in interpretation of the specification. When an artificial language is used, the precise meaning of each concept may be fixed and documented. This leads to unambiguous interpretation of specifications using this language. Finally, an artificial language enables the designer to constrain the semantics (and thus the requirements statements) to a single, appropriate level of detail. Therefore, requirements in RSL can be (and are) limited to statements of true requirements that are testable in the software and are not overconstraining.

RSL is the primary means that a requirements engineer uses to communicate with REVS, so its characteristics can easily spell the success or failure of SREM. Therefore, we devoted considerable effort to RSL design and provide some of the important details of that design below.

Flows

The most obvious way to state system requirements for software is to describe the operations that each software module shall perform; these requirements differ little from program specifications. This type of "requirements specification" contains little information about the truly required sequence of processing, or even about communication between designed modules. Stating the requirements (even if they are at the correct level) without indicating the required sequence ignores problems in BMD systems and most other process control systems. The basic approach therefore must be statement of required operations as flows through the system. This orientation is facilitated by the stimulus-response nature of process control systems; each flow originates with a stimulus and continues to the final response. Specifying requirements in this fashion makes explicit the sequences of processing required. It also provides for direct testability of the requirements; the software may be tested to see if it provides the responses which were specified when given the associated stimulus.

Flows through the system are specified in RSL as requirements networks, also called R_NET's. In addition to enabling conditions, information requirements, and other material in R_NET's, they have flow structures consisting of nodes, which specify processing operations, and the arcs which connect them. The basic nodes include ALPHA's, which are the specifications of functional processing steps, and SUBNET's, which are specifications of processing flows at a lower level in the hierarchy. In essence, the SUBNET is an ALPHA which has been expanded to include internal details of the processing. All of the basic nodes are single-entry, single-exit. In addition to the simple sequential flow which may be represented by connecting this type of node, more complex flow situations are expressible in RSL by the use of structured nodes which fan-in and fan-out to specify different processing paths; the structured nodes are the AND, OR, and FOR EACH.

The AND node is the first of three structured nodes. The meaning of an AND structure, which contains several paths, is that these paths are mutually order-independent. The processes on parallel paths may be executed in any order, or even in parallel. The fan-in at the end of the AND structure is a synchronization point; all of the parallel paths must be completed before any of the processes following the rejoin are performed.

The second structured node, the OR node, also has several paths. A condition is attached to each path at an OR-type fan-out. The one path with a true condition is processed; the others are ignored. Each OR node must have an OTHERWISE path that is processed if none of the conditions are true. In case more than one condition is true, the first path (as indicated by either implicit or explicit ordering) is processed.

The final structure type is the FOR EACH. This structure contains only one path; this path is processed once for each element of a set of data-processing system entities that is currently present. For example, a requirement might state that proximity of interceptors be determined FOR EACH reentry object. The FOR EACH, therefore, specifies that a particular process or set of processes is to be iterated upon without sequential implications.

The syntax of the structures in R_NET's makes the basic nodes and the several types of fan-out nodes explicit. The fan-in nodes and all of the arcs are visible only by implication. An example of such a structure is the one shown in Fig. 2 and the syntax to represent it is shown in Fig. 3.

The syntax of structures in R_NET's is similar to the syntax of many structured programming languages and is designed to achieve the same effect; that is, enforcing a discipline on the user. Through the use of a fixed set of flow primitives, flow

structures which are ambiguous or unclear are precluded from appearing on R_NET's. This helps the requirements engineer himself where he is vague or ambiguous, aids in the communication of requirements between requirements engineers, and permits a highly automated analysis of the requirements by the tools.

75

Extensions

RSL is an extensible language in order to permit the inclusion of new concepts that may be needed for future requirements. Typically, BMD software must perform guidance (largely an analytical problem), must allocate resources (a heuristic problem), and must interact with all other parts of the BMD system (a coordination problem). The processing demands of such systems are so large that the state of the art (in hardware, software, algorithm concepts, etc.) is regularly stretched by the design of the data processing system, and the art advances quite rapidly. Combined with these difficulties for providing a language to state data processing requirements is our inability to know about future potential developments in other parts of a BMD system (radar, kill mechanism, etc.) which may require special interfaces or processing techniques. Any language with fixed concepts would quickly become inappropriate for stating such requirements because it would lack at least one needed concept. Therefore, RSL needed to be extensible.

To facilitate the implementation of extensibility at the concept level and to provide a clear framework for the concepts, the underlying architecture of RSL has been kept very simple. There are four primitives, outlined below, which form the foundation of the language.

1) *Elements*: Elements in RSL correspond roughly to nouns in English. The element types are simply standard prototypes which are used to describe the properties possessed by each element of the type. Some examples of standard element types in RSL are ALPHA (the class of functional processing steps), DATA (the class of conceptual pieces of data necessary in the system), and R_NET (the class of processing flow specifications).

2) *Relationships*: The relation (or relationship) in RSL may be compared with an English verb. More properly, it corresponds to the mathematical definition of a binary relation, a statement of an association of some type between two elements. The RSL relation is noncommutative; it has a subject element and an object which are distinct. However, there exists a complementary relationship for each specified relationship which is the converse of that specified relationship. DATA being INPUT TO an ALPHA is one of the relationships in RSL; the complementary relationship says that the ALPHA INPUTS the DATA.

3) *Attributes*: Attributes are modifiers of elements somewhat in the manner of adjectives in English; they formalize important properties of the elements. Each attribute has associated with it a set of values which may be mnemonic names, numbers, or text strings. Each particular element may have only one of these values for any attribute. An attribute may pertain to all element types or may be restricted to be used with only a certain set of types. An example of an attribute is INITIAL_VALUE which is applicable to elements of type DATA. It has values which specify what the initial value for the data item must be in the implemented software and for simulations.

4) *Structures*: The final RSL primitive is the structure, the RSL representation of the flow model mentioned earlier. It is a mapping of a two-dimensional graph structure into a one-dimensional stream of computer input. It models the flows through the functional processing steps (ALPHA's) or the flows between places where accuracy or timing requirements are stated (VALIDATION_POINT's).

All concepts in RSL are expressed in terms of these four primitives. The structures are not extensible; this preserves the necessary discipline for flow descriptions. On the other hand, new types of elements, relationships, and attributes may be added to the language at will to express new concepts. In fact, all concepts (both new and old) are defined as extensions.

As an example of using the extension capability, a requirement often involves information entering the data processing system from outside hardware (radars, operator consoles, etc.). The port into the data processing system is embodied in a type of element called an INPUT_INTERFACE. The complex of information which it handles is called a MESSAGE; the INPUT_INTERFACE then PASSES the MESSAGE into the data processing system. The definitions of these two types of elements and their connecting relationship in RSL are as follows:

```
DEFINE ELEMENT_TYPE: INPUT_INTERFACE
                     ("A port between the data processing system and the rest of the
                     system which accepts data from another part of the system").

DEFINE ELEMENT_TYPE: MESSAGE
                     ("An aggregation of DATA and FILES that PASS through an interface as a logical unit").

DEFINE RELATIONSHIP: PASSES
                     ("An INPUT_INTERFACE "PASSES" a logical aggregation of data called a MESSAGE from the outside system into the data processing system").

    COMPLEMENTARY RELATIONSHIP: PASSED ("BY").

    SUBJECT: INPUT_INTERFACE.

    OBJECT: MESSAGE.
```

After these definitions are processed by the RSL translator and entered into the ASSM, they are available for use by any requirements engineer working with that data base. Of course, to make these particular definitions useful, additional relationships must be defined; of particular importance is the relationship which associates particular items of DATA with the MESSAGE.

With an extension mechanism this powerful, it would be a tempting proposition to provide only this mechanism so that the requirements engineer would have a means to provide his own concepts. This, however, would demand that the engineer design his own language for requirements statement as well as

engineering the requirements themselves. Therefore, we have provided a *core* set of concepts that appears to be needed with great regularity, and have also provided the mechanism for extending the concepts as needed in particular problems. The examples of extensions given above are a part of this core set along with 19 other element types, 20 relationships, and 20 attributes.

Core Concepts

The following examples show the use of the core concepts to define requirements. They include the definitions of two required processing steps (ALPHA's), of a required item of information (DATA) which is to be OUTPUT FROM one of the ALPHA's, and of a statement from another document which necessitates the inclusion of several items in the requirements (ORIGINATING_REQUIREMENT).

```
ALPHA: EXTRACT_MEASUREMENT.
    INPUTS: CORRELATED_RETURN.
    OUTPUTS: VALID_RETURN, MEASUREMENT.
    DESCRIPTION: "DOES RANGE SELECTION PER CISS
        REFERENCE 2-7".
    ENTERED_BY: "M. RICHTER".

ALPHA: DETERMINE_IF_REDUNDANT.
    INPUTS: CORRELATED_RETURN.
    OUTPUTS: REDUNDANT_IMAGE.
    DESCRIPTION: "THE IMAGE OF THE RADAR RETURN
        IS ANALYZED TO DETERMINE IF IT IS REDUN-
        DANT WITH ANOTHER IMAGE".
    ENTERED_BY: "F. BURNS".

DATA: MEASUREMENT.
    INCLUDES: RANGE_MARK_TIME, AMPLITUDE,
        RANGE_VARIANCE, RD_VARIANCE,
        R_AND_RD_CORRELATION.
    OUTPUT FROM: ALPHA EXTRACT_MEASUREMENT.
    DESCRIPTION: "THIS IS THE ESSENCE OF THE INFOR-
        MATION IN THE RETURN".
    ENTERED_BY: "F. BURNS".

ORIGINATING_REQUIREMENT:
        DPSPR_3_2_2_A_FUNCTIONAL.
    DESCRIPTION: "ACTION: SEND RADAR ORDER IN-
        FORMATION: INFORMATION: RADAR ORDER,
        IMAGE (REDUNDANT)".
    TRACES TO: ALPHA COMMAND_PULSES
               ALPHA DETERMINE_IF_REDUNDANT
               MESSAGE RADAR_ORDER_MESSAGE
               DATA REDUNDANT_IMAGE
               ENTITY_CLASS IMAGE.
    ENTERED_BY: "T. E. BELL".
```

The Translator

The RSL translator is a component of REVS; its purpose is to analyze the RSL statements which are input to it and to make entries in the ASSM corresponding to the meaning of the statements. It does this by extracting the RSL primitives (elements, relationships, attributes, and structures) which exist in the input statements, and by mapping them to constructs in the ASSM. The translator also processes modifications and deletions from the data base which are commanded by RSL statements specifying changes to already-existing instances in the ASSM. This is done in a manner similar to that used in processing the additions to the ASSM; that is, the primitives are extracted and the referenced construct is modified or deleted. For all types of input processing, the translator references the ASSM to do simple consistency checks on the input. This prevents the occurrence of disastrous errors such as the introduction of an element with the same name as a previously existing element or an instance of a relationship which is tied to an illegal type of element. Besides providing a measure of protection for the data base, this type of checking catches, at an early stage, some of the simple types of inconsistencies that are often found in requirements specifications.

In addition to processing requirements statements given in terms of the core set of extensions, the translator accepts further extensions and enters them into the ASSM. Obviously, the extensions must be treated with even more care than the requirements statements since the deletion of, say, a previously legal element type may invalidate a large segment of the requirements. On the other hand, adding new concepts in response to each individual's desires is a pernicious practice. For these reasons, a lock mechanism has been built into the translator to enable it to reject any extensions while locked. This allows the management of a project to control the use of the extensions in their project and to enforce a disciplined use of the power of RSL.

The translator has been implemented using a compiler writing system [16]; this adds additional flexibility to the language. Changes to the syntax or the primitives of the language, which would be unthinkably expensive with a hand-built translator, may be accomplished with relatively small changes to the inputs of the compiler writing system. Thus, for example, if evolutions in the Software Requirements Engineering Methodology introduce a type of flow structure which does not currently exist in the language, the structure portion of RSL may be changed through the use of the compiler writing system. Of course, the impact of changes of this magnitude on the ASSM and tools are significant. Therefore, the use of this final level of flexibility is tightly controlled.

V. The Abstract System Semantic Model

The RSL statements that an engineer inputs to the Requirements Engineering and Validation System (REVS) are analyzed, and a representation of the information is put into a special data base. This data base is called the "Abstract System Semantic Model" (ASSM) because it maintains information about the required data processing system (RSL semantics) in an abstract, relational model. Each statement is checked for syntactic and elementary semantic correctness prior to being put into the ASSM. Therefore, a number of different tools can easily access the data without the need to check for these types of correctness with each access.

In addition to maintaining information about the requirements, the ASSM maintains the concepts used to express the requirements. This means that all extensions, both the core concepts and the additions and modifications of specific projects, are in the ASSM. This allows the RSL translator to

process extensions in a manner similar to that in which it processes the requirements statements. The modified concepts are then available for use as soon as they are entered.

Information in the ASSM is not simply a collection of text. Instead of this, it exists as a relational model of the information contained in the RSL statements. In this model, elements are represented by nodes (records) in the data base, and relationships are represented as connections between the nodes. Attributes and their values then consist of a node for the value and a connection to the element node with which the attribute is associated. The structures are expanded to the graph which they represent. This type of representation facilitates retrieval of information from the data base in terms of queries about the relationships between elements; complex combinations of relationships can be traced simply by following the proper connections in the ASSM. In order to support extensions of the language, another level of model exists. At this level prototypes for each type of element, relationship, or attribute may legally be represented. Adding concepts then translates to adding new prototypes to the model. Each instance of an element, relationship, or attribute is linked to its prototype. This facilitates concept-oriented retrieval operations such as "Find all elements of type DATA which are not INPUT TO anything" as well as facilitating extensibility.

In addition to providing a means to enhance the efficiency of processing, the ASSM provides a central repository for all information about the system data processing requirements. In the environment of a large BMD system, this type of centralization is necessary since many individuals are continually adding, deleting, and changing information about requirements for the data processing system. The ASSM provides a means for all of them to work with the same base of current information; they can find out about the effects of their work on other requirements engineers, the characteristics of parts of the system that other people are defining, and the current status of their own work. The centralization allows both the requirements engineers and the analysis tools to work from a common baseline and enables implementation of management controls on changes to this baseline.

The function of the ASSM as a central repository for all information is crucial to both the extensibility and disciplined approach aspects of SREP. Extensibility of the language at the level of adding, deleting, or modifying concepts would be nearly impossible if those concepts were spread throughout the system. In addition, the residence of all information about both the software requirements and the concepts used for describing them in the ASSM makes it possible to take a modular and extendable approach to the tools. It also permits the imposition of configuration management controls in an efficient manner since blocking of modifications to the ASSM freezes the configuration. Finally, the analysis tools can easily scan through the data base to check for consistency, a task which would be extremely difficult without access to the description of the entire system at once.

VI. Automated Tools

For a large software system, many people develop requirements for different segments of the system; using SREM, each person develops RSL descriptions of the requirements for his particular part of the system. REVS assists in this activity and also provides mechanisms for imposing discipline and control on the requirements and the requirements engineering process. This is accomplished by the third segment of REVS, the automated tools. The requirements engineer uses the tools to identify those areas which need further resolution, to aid him in resolving problems, and to evaluate his inputs. At various milestones in the development, the tools are also used to evaluate the entire system. Typical requirements engineering efforts will have several iterations of this type, with the tools being used at each stage to show areas which need further work.

The baseline set of tools in REVS contains flow-oriented analysis and validation aids for verifying the completeness, consistency, and correctness of the specified requirements. A generalized extraction and reporting system provides additional analysis and status information. While the flow-oriented analysis aids are not easily extensible, the flexible extraction system is a powerful extension capability adaptable both by the engineer to evaluate his requirements, and by management to maintain visibility and control.

Flow Orientation

The stating of requirements in RSL is based on identifying and relating other requirements information to specified functional flows of processing steps. In REVS, we have implemented tools to provide graphics input/output, to perform static analysis, and to create simulators based on this flow approach. These tools aid the engineer in developing the flows and in validating the consistency, completeness, and correctness of the specified flow structures and their related requirements.

Interactive Graphics: The interactive R_NET generation tool provides graphics capabilities for users of REVS. Through this facility, the requirements engineer may input, modify, or display R_NET's. It also provides an alternative to the RSL translator for specification of the flow portion of the requirements. Using this tool, the user may even develop a graphic representation of an R_NET previously entered in RSL. These flow diagrams (Fig. 2) are inherently two dimensional, so their display on a graphics system provides a more-easily understood representation than the (one-dimensional) language (Fig. 3). Through the use of the ASSM, the user may work with either the graphic or RSL language representation of R_NET's; they are completely interchangeable.

The interactive R_NET generation facility possesses full editing capabilities. The user may input an R_NET "from scratch" or he may modify one previously entered. At the conclusion of the editing session, the new R_NET replaces the old one (if any) in the ASSM. An alphanumeric keyboard and a trackball-driven cursor are the means by which the user communicates with the graphics system. He can select any of a series of functions from a menu of available functions. The editing functions provide means to position, connect, and delete nodes, to move them, to disconnect them from other nodes, and to enter or change their associated names and commentary. Menu selection and screen positioning are done using the trackball; names and commentary are entered through the keyboard. The size of an R_NET is not

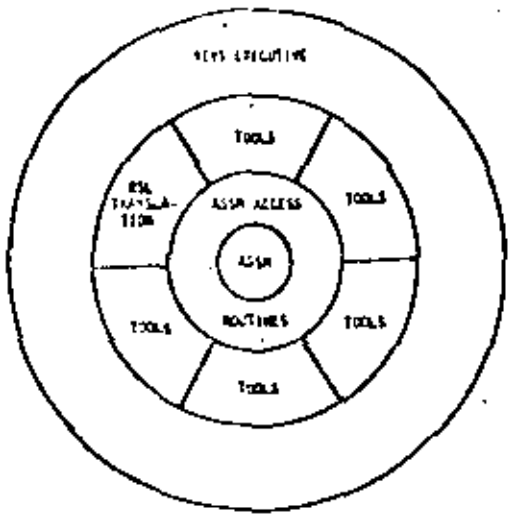


Fig. 4. REVS layered design.

to RSL and insulate the tools from the detailed organization of the data base. The ASSM can be physically restructured without impacting the tools.

The REVS executive invokes the automated tools based on command from the user. A new tool may be easily incorporated into REVS as an integral part by modifying the executive to recognize a new command and invoke the added tool.

Special Reports: The structure of REVS easily accommodates extensions to the baseline tools. However, adding a new tool each time a requirements engineer needs a special report or analysis is costly and does not allow timely responsiveness to needs. REVS alleviates this problem by providing the requirements engineer an extensible tool (the extractor) to produce specialized reports. The user completely controls the scope of the analysis and content of his reports; he is not burdened with the details of specifying the format nor with looking at tabular forms to extract needed information.

This flexibility of extraction from the data base is provided by the generalized extractor system. Using this system, the user can subset the elements in the ASSM based on some condition (or combination of conditions) and display the elements of the subset with any appended information he selects. Output is in a standardized form compatible with the RSL input form. Prepositions and additional punctuation are added so that a formal documentation of the requirements can be generated in an order standardized to the needs of a particular application. Being standardized RSL, outputs of the extractor can be easily correlated with the RSL inputs and the engineer has to deal with only one requirements format.

Information to be retrieved is identified in terms of RSL concepts. For example, if the user wants a report listing all DATA elements which are not INPUT TO any ALPHA (processing step), he enters the following commands.

```
SET A = DATA THAT IS NOT INPUT.
LIST A.
```

By combining sets in various ways, he can detect the absence and presence of data, trace references on the structures, and analyze interrelationships established in the ASSM. In analyzing user requests and extracting information from the ASSM, the extractor system uses the definition of the concepts contained in the ASSM. Thus, as RSL is extended, extensions and their use in the requirements are immediately available.

The extractor system can be used both for *ad hoc* inquiries and for routinely generated extractor special reports. Both the requirements engineer and management may predefine reports and enter the requests to the extractor as needed. This allows the requirements engineer to establish a repertoire of situation-specific consistency and completeness checks, and to perform automatic regressive testing. Managers can periodically request reports on the status of the ASSM to impose standards and control the development of requirements. For example, RSL supports the traceability of software requirements to system requirements to permit rapid, economical, and comprehensive change control. Using the generalized extractor system, a report can be produced at any time to show, for example, which requirements have no documented traceability and are therefore suspect.

VII. Conclusions

The REVS software is now operational on the Texas Instruments Advanced Scientific Computer at the BMDATC Advanced Research Center in Huntsville, AL. The software executes in the manner described in this paper. This fact, though comforting, is hardly adequate for a justification of the effort to produce the system. If we were to conclude this paper with nothing more than the proud assurance that the system does not abnormally terminate, we would fall into a category of system builders that B. Boehm has called "computer basket weavers."

Referring to people in this category, Boehm stated "A basket weaver has a very difficult job. He must plan his basket very carefully and he puts a lot of loving care into it; he builds it, studies it from various angles, discusses it with other basket weavers, and then goes off to build another basket. Very rarely though does he go out and sample users to find out whether they are interested in baskets with handles or with several compartments rather than one compartment, and the like. And, unless something changes considerably in computing, it will remain a kind of computer basket weaving" [20]. Boehm is criticizing computer science researchers for not doing adequate requirements analyses and for not checking the degree to which the systems fulfill the requirements after their implementation. A system purporting to improve requirements engineering certainly has a special responsibility to identify its requirements and then to check the degree of fulfillment.

The central nature of RSL led us to concentrate on its requirements in our early development—and to document the requirements so that we could track our progress. The material in Section II of this paper is a summary of that document after we had expanded our requirements to include the software in REVS. Merely having some set of requirements, of course, does not prove anything; how well does the system actually satisfy the users' needs?

Some of the test cases used to evaluate requirement satisfac-

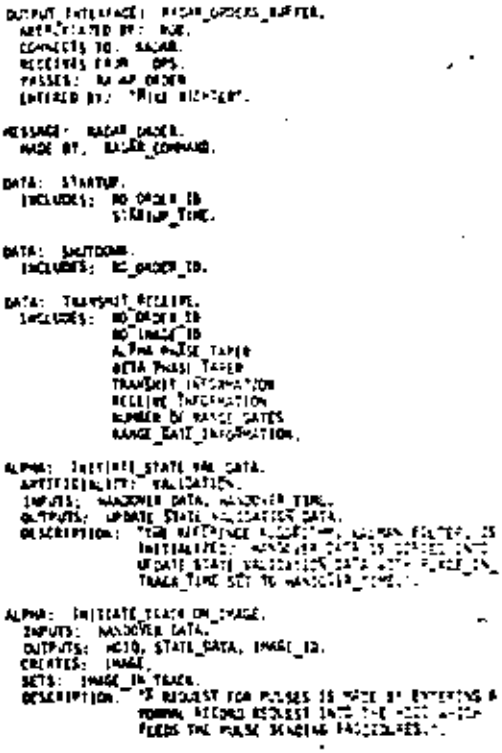


Fig. 5. RSL from BMD track loop.

tion using SREM are noted in Alford's paper [13]. Most of those, however, involved other parts of SREM than RSL and

S. The cases noted below were particularly important in ...ing out whether users were interested in the "basket" that we had woven with RSL and REVS.

One of our first test cases involved restating into RSL a particularly involved part of the requirements for a medical information system. We hoped that the RSL statements would be clear enough that the user (a professional data processing specialist deeply involved in medical systems) would find graphical representations superfluous. Instead, we found that the graphics were really necessary for him to understand the flows through his system. When he used both RSL and graphics, he found five critical problems in the English requirements statements. One of these problems, for example, involved requiring a physician to check the same medical record five times when a single check would have sufficed. Implementing this system would have required changes in (and wastage of) physicians' time—if they had used the system at all! Clearly, our software system needs graphical output if it is to handle situations like this.

We were particularly interested in the medical system for this evaluation because its flows were far shorter than in a BMD system. Therefore, any indication of confusion in the absence of graphics would be magnified in the BMD situation. With the results described above, we were certain that graphics was required for our "basket" to satisfy the users' needs.

A later test case involved having BMD data processing engineers develop the requirements for an actual BMD function (tracking). In this case the RSL translator was implemented to the point that we could input the RSL statements and have them checked for consistency. Our experience in this case indicated that some of the concepts in the core set of RSL were inappropriately conceived; they were unambiguous, but they were not sufficient to express all of the requirements.

This BMD case resulted in revisions to RSL concepts that were incorporated into the baseline version of the language description. We found that the extensibility features of RSL worked, and we were able to revise the language without changing the translator. Thus, we found that we needed some features of our "basket" revised (the core concepts) but that other features were solid (the extensibility).

After the revisions to the core concepts, another BMD data processing engineer (one new to the project and not particularly sympathetic to many of its conclusions) started over on track-loop (the tracking function) using the revised RSL. He found a few further revisions that he desired in a concept area not fully explored previously. Again, concept changes were handled with the extension capability. The "basket" seemed to be getting quite close to the users' needs for stating requirements.

One type of user is the person writing the requirements in RSL, but another is the person reading the resultant statements. The information needed by the reader must be in the RSL statements written by the requirements engineer or the system is, at least, severely crippled. RSL text is rather cryptic, and readers expressed concern about whether adequate information existed in its easily written form. We performed another test case to evaluate whether our "basket" fulfilled the readers' information needs.

The RSL from the track-loop test case had given readers the impression that inadequate information existed there, so we used the actual RSL from that test case (an extract appears in Fig. 5). Then we modified the RSL with phrases that substituted for the standard RSL element, relationship, and attribute names. Finally, we had the material typed in a conventional

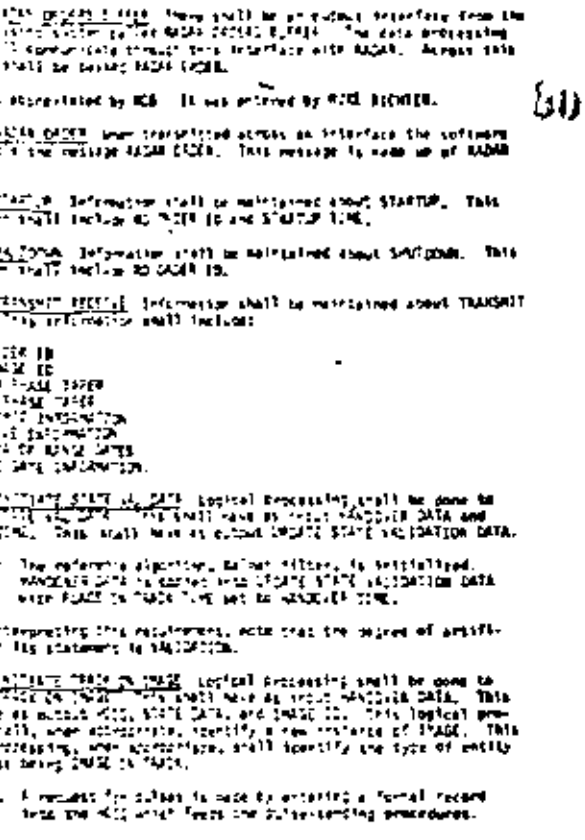


Fig. 6. "Conventional" format for track loop.

format and added the usual paragraph numbers. All of this added no new information: all the changes could easily have been done by a computer: in fact, we documented our algorithms to be sure that we exercised no discretion in making the changes.

We tested the sufficiency of the RSL that we had put into a familiar format (with lots of redundant material) by presenting it to readers unaware of its origin. Without exception they thought the text (an extract is shown in Fig. 6) was from a real, normal specification. In fact they sometimes needed to see the original RSL to be convinced that it was merely the same material reformatted. The needed information, at least at the paragraph level, is contained in RSL statements; perhaps an automated capability to produce more familiar-looking output should be included in future versions of REVS to make our "basket" as useful as possible.

Our evaluation of RSL and REVS is clearly not yet complete since users have not yet employed the final versions of the software. We will continue our evaluation of SREM utility to ensure that we have actually produced an extensible system that encourages disciplined thinking in engineering correct, complete, meaningful software requirements. Our evaluations to date give every indication that RSL and REVS actually satisfy a critical need in furthering the development of large-scale software.

References

[1] J. Goldberg, Ed., *Proc. Symp. on the High Cost of Software*, Naval Postgrad. School, Monterey, CA, Sept. 17–19, 1973.

[2] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, 19, pp. 48–59, May 1973.

[3] J. D. McGonacle, "A study of a software development project," James P. Anderson & Co., Fort Washington, PA, Sept. 1973.

[4] J. C. Bartlett *et al.*, "Software validation study," LOGICON Rep. DS-72210-R1330, NTIS Doc. AD-759 263, LOGICON, Inc., San Pedro, CA, Mar. 1973.

[5] T. A. Thayer, "Understanding software through empirical reliability analysis," in *Proc. 1975 Nat. Comput. Conf.*, June 1975, pp. 335–341.

[6] T. A. Thayer *et al.*, "Software reliability study: Final technical report," study performed by TRW Defense and Space Systems Group for the Air Force Systems Command's Rome Air Development Center, Griffiss Air Base, Rome, NY, Feb. 27, 1976.

[7] T. E. Bell and T. A. Thayer, "Software requirements: Are they really a problem?," in *Proc. 2nd Int. Software Eng. Conf.*, Oct. 1976.

[8] A. Asch, D. W. Kelliher, J. P. Locher, III, and T. Connors, "DOD weapon system software acquisition and management study, volume I, MITRE findings and recommendations," MITRE Tech. Rep. MTR-6908, The MITRE Corp., McLean, VA, May 1975.

[9] A. Kossiakoff, T. P. Sleight, E. C. Prettyman, J. M. Park, and P. L. Hazan, "DOD weapon systems software management study," APL/JHU SR 75-3, The Johns Hopkins Univ. Appl. Phys. Lab., Silver Spring, MD, June 1975.

[10] BMD Advanced Technology Center, "BMDATC software development system," vol. I and II, Ballistic Missile Defense Advanced Technology Center, Huntsville, AL, July 1975.

[11] D. Teichroew, E. A. Hershey, and M. J. Bastarache, "An introduction to PSL/PSA," ISDOS Working Paper 86, Univ. Michigan, Ann Arbor, Mar. 1974.

[12] E. A. Hershey, "A data base management system for PSA based on DBTG 71," ISDOS Working Paper 88, Univ. Michigan, Ann Arbor, Sept. 1973.

[13] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," this issue, pp. 60–69.

[14] M. W. Alford and I. F. Burns, "R-nets: A graph model for real-time software requirements," in *Proc. Symp. on Comput. Software Eng., MRI Symp. Ser.*, vol. XXIV. Brooklyn, NY: Polytechnic Press, to be published.

[15] T. E. Bell and D. C. Bixler, "A flow-oriented requirements statement language," in *Proc. Symp. on Comput. Software Eng., MRI Symp. Ser.*, vol. XXIV. Brooklyn, NY: Polytechnic Press, to be published. (Also in TRW Software Series, TRW-SS-76-01.)

[16] O. Lecarme and G. V. Bochmann, "A (truly) usable and portable translator writing system," in J. L. Rosenfeld, Ed., *Information Processing 74*. Amsterdam: North-Holland, 1974.

[17] F. J. Mullin, "Software test tools," in *Proc. TRW Symp. on Reliable, Cost Effective, Secure Software*, TRW Software Ser. Rep. TRW-SS-74-14, pp. 6-47–6-48, Mar. 1974.

[18] K. Jensen and N. Wirth, "PASCAL: User manual and report," *Lecture Notes in Computer Science*, vol. 18. Berlin: Springer-Verlag, 1974.

[19] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 137–147, Mar. 1976.

[20] B. W. Boehm, "Command/control requirements for future Air Force systems," in *Multi-Access Computing: Modern Research and Requirements*, Rochelle Park, NJ: Hayden, 1974, pp. 17–29.

Thomas E. Bell (S'62-M'63) was born in Phoenix, AZ, in 1940. After attending Harvey Mudd College, Claremont, CA, for 2½ years, he received the B.S. degree in applied physics from the University of California, Los Angeles, in 1963. His M.B.A. and Ph.D. degrees were from UCLA's Graduate School of Business in 1964 and 1968, respectively.

He was a member of the Rand Corporation's professional staff from 1967 through 19[illegible]. During this time he performed research in appropriate techniques to use computer graphics (both interactive and off-line) in human problem solving. However, his main emphasis was on the development of techniques for computer performance evaluation. In recognition of his contributions both during this period and subsequently, he was awarded the A. A. Michaelson Award in 1975 for out-

standing personal contributions to computer performance evaluation. In 1974 he joined the Software Research and Technology Staff of TRW. His efforts were devoted to improving the development of software, both through research and application, in the environment of large, complex systems. Analysis of empirical data, development of new techniques, and design of a new software requirements statement language were emphasized. He is currently a Manager at Peat, Marwick, Mitchell & Copartners, in the Management Consulting Department. He is responsible for practice development and application in the area of productivity enhancement techniques for computers and the humans using and operating them.

Dr. Bell is a member of the Association for Computing Machinery, the IEEE Computer Society, the Institute for Management Sciences, and the Computer Measurement Group. He is the author of over 30 papers in computer science.

David C. Bixler was born in Youngstown, OH, on August 1, 1949. He received the B.S. degree in chemical engineering in 1971 and the M.S. degree in computer science in 1973, both from Michigan State University, East Lansing.

Since 1973 he has been associated with the Applied Software Laboratory of TRW Defense and Space Systems Group, Redondo Beach, CA, where he has worked in the areas of software requirements engineering, computer description languages, and computer emulations. His primary areas of interest are the semantics of programming languages and verification techniques for computer programs.

Mr. Bixler is a member of Tau Beta Pi and the Association for Computing Machinery.

81

Margaret E. Dyer received the B.S. and M.S. degrees in mathematics in 1964 and 1965, respectively, from the University of Michigan, Ann Arbor.

From 1965 to 1973 she was associated with Teledyne Brown Engineering where her primary responsibilities were the development of a conversational language processor, the development and application of simulations, and the comparative evaluation of fourth generation data processors. In 1973 she became a member of the Technical Staff of the MITRE Corporation. At MITRE and, subsequently, with TRW Defense and Space Systems Group, she has been involved with the development of advanced techniques for the development of software for large-scale, real-time control systems. Since joining TRW in 1974 she has concentrated on research and development of advanced software requirements engineering languages and automated tools. As Manager of the REVS Software and Language Development at the TRW Huntsville Facility, she is responsible for the design and development of the Requirements Engineering and Validation System.

THE *EVEREST* OF SOFTWARE

Lewis M. Branscomb

Vice President and Chief Scientist, IBM Corporation, Armonk, NY

Much has been written about the tremendous advances of computing over the past few decades. In fact, when we computerniks get together, we generally pat each other on the back and tell ourselves how great we are. I, myself, am quite guilty of quoting frequently the famous statement by John Pierce, who said: "After twenty-five years of extraordinary progress, the computer industry is ready to enter its infancy."

Before I comment on this statement in detail, let me correct an historical inaccuracy. It is not true that computers have been with us for only twenty-five years. In fact, the earliest recorded reference to computers was made by Jonathan Swift in his "A Voyage to Laputa," Chapter 5, where he describes a visit of Gulliver to the Academy of Laputa. This passage describes a device invented by the Professor of Speculative Learning, which can be described as nothing less than a computer, and, in fact, contains many of the elements of some computer science projects of our days.

The computer of Laputa was a "Frame" that contained "· · · all the words of their language in their several Moods, Tenses, and Declensions, but without any Order · · ·." These words were cranked out by the Professor's pupils, and any strings of them that could compose a piece of a sentence were put aside. Thus the Professor expected to create "a complete Body of all Arts and Sciences; which however might be still improved, and much expedited, if the Publick would raise a Fund for making and employing five hundred such Frames · · ·." As you can see, this passage contains not only a fairly technical discussion of a modern Monte Carlo text generation project, but, also, the usual appeal for government support.

Let me now analyze the elements of a computing process. According to Harlan Mills, a computing process is nothing more nor less than the operation of a multiprocessing system. Even the simplest computing operation involves at least two processors: a human being; and a computer. In order to make these two processors work together, we need two other elements: a users' language which interfaces with the human being; and a software package which interfaces with the computer.

Let us now go over these four elements of a computing process and see how well John Pierce's statement reflects reality; that is, how well we stand today with respect to the development of these four elements, and where we are going from here. Let's take the human being first. I think it is fair to say that there is no reason for great concern for the technological development of that element. It has been engineered for many centuries, and its manufacturing process has been perfected to a great extent. I would be perfectly willing to say more about this subject, but I understand it is outside the scope of this meeting.

The users' manual is a puzzling subject to me. As near as I can make it, there are two kinds of people in this world, the writers of users' manuals and the readers of users' manuals. The two groups work on completely different planes of understanding very

---

Address presented at the banquet following the opening day of the symposium on Computer [illegible]are Engineering, Polytechnic Institute of New York, April 20-[illegible]2

time I open a users' manual, I remember the following opening sentence in a book by A. A. Blasov of Moscow University: "The purpose of the present course is the deepening and development of difficulties underlying contemporary theory." Nevertheless, I feel that we are only a couple of mutations away from developing a breed of computer specialists who can understand users' manuals, so I will not worry about this particular topic.

The third element of the process is the computer itself. There we have absolutely nothing to worry about. As everyone knows, the advances in computer hardware have been so phenomenal that even the pioneers in the computing industry are amazed at how far we have gone during the past twenty-five years. This is, of course, exactly what John Pierce meant by his famous statement. Progress in computer hardware, with respect to cost, speed of computations, and decrease in size, is measured by several orders of magnitude, and the most dramatic is the 40% compound annual rate of reduction in the unit cost of memory and storage. This has permitted all kinds of shortcomings in software efficiency to be swept under the rug of expanding memory.

So now, let us take a look at the fourth element of the process, the software. There we seem to be in trouble. We all know that a science of software has yet to be developed. Software design is still largely an art, and you good people are the struggling practitioners of this noble art. I know how difficult your struggle is, and I want to tell you that I am very sympathetic with your problems. Of course, for me to display my sympathy with your problems is very much like the practice of certain Greek peasants of lying in bed in sympathy for their wives who are convalescing from childbirth.

But what is really wrong with the software process? One of the problems, of course, is the increasing complexity of software, particularly systems programs. The size of systems programs was measured in a few thousand instructions in the early days of computing, but it is measured in millions of instructions today, in some cases. What is more, we seem to be unable to develop building blocks of software, analogous to the building blocks that we have developed for hardware. In fact, it is worse than that. It has been said that every systems program is done by an amateur, because everyone who has prepared a systems program once doesn't want to do it ever again.

Everyone who has given any thought to the problem of software comes to the conclusion that we *must* develop a design methodology for software. I wish I could have stood before you today and given you a blueprint for the development of this design methodology. I cannot give you that, but I can try to identify some principles that we may use in scaling this seemingly unreachable height, the *Everest* of software. First of all, I think we must do away with our sometimes excessive preoccupation with efficient use of the hardware. This may sound like a self-serving statement, in view of the nature of my employment, but it is not. I assure you that I am speaking at this moment as a user of a computer, and not as a seller of hardware. There *must* be a balanced consideration of questions of efficiency.

Let me illustrate where efficiency may sometimes lead you by reading a passage from a book titled "A Random Walk In Science," which contains a report by an anonymous author of a visit by a team of efficiency experts to the Royal Festival Hall:

> "For considerable periods the four oboe players had nothing to do. Their numbers should be reduced, and the work spread more evenly over the whole of the concert, thus eliminating peaks of activity. ··· All the twelve first violins were playing identical notes. This seems unnecessary multiplication. The staff of this section should be drastically cut; if a large volume of sound is required, it could be obtained by means of electronic amplifiers. ··· Much effort was absorbed in the playing of demisemiquavers. This seems an excessive refinement. It is recommended that all notes should be rounded up to the nearest semiquaver. If this were done it would be possible to use trainees and lower grade operatives more extensively. ··· There seems to be too much repetition of some musical passages. Scores should be drastically pruned. No useful purpose is served by repeating on the horns a passage which has already been handled by the strings. It is estimated that if all redundant passages were eliminated the whole concert time of two hours could be reduced to twenty minutes, and there would be no need for an interval."

On a more serious vein, efficient use of hardware must be balanced against efficient use of the programmers who write software. Sometimes, I feel that we have been conditioned from the days when computers were very expensive tools that had to be extremely well utilized. This is a well-known economic principle, that is not only appropriate for the use of computers. A piece of machinery must be utilized a lot better in India where its relative value with respect to human labor is much higher than it is in this country. The same piece of machinery can be *burned* in this country in order to save a few man-hours which are expensive.

I think there is a certain lag in our awareness of the decreasing costs of hardware as a percentage of the total cost of the computing process. If we look at any data, we come to the conclusion that the relative cost of hardware, which started at something like 90% of the total cost of computing in the early days of computing, is now well below 50%, and slated to be less than 10% by the middle 1980s. When the software cost becomes nine times the hardware cost, simple arithmetic will tell you that you can afford to double the cost of hardware if you can save anything more than one-ninth of your software expenditures. If you want a more sophisticated economic statement, it must be: "The code efficiency must be at a level such that the cost for a marginal increase of efficiency must be equal to the corresponding marginal decrease of hardware costs." I have a nagging suspicion that we are not running our computing business according to this principle today. The problem has its roots in the lack of a methodology for performance specification and evaluation—a way to put together the appropriate mix of hardware and software to meet performance requirements.

I have stated an economic principle in balancing software development costs against hardware costs, but I am well aware that even the best economic principle does not by itself solve the software design problems. I have mentioned, earlier in my talk, the desire to have modularity in software, which would permit building a large system out of building blocks. But modularity implies stabilized interfaces. And the rate of evolution of data processing technology—the shift from batch processing to time-sharing, and from remote job entry to on-line interaction, plus networking and distributed processing, makes this stabilization very difficult at this stage of the computing industry.

Undoubtedly, as our industry matures, some stabilization in software design interfaces will be implemented, which will hopefully accomplish two things. First, it will make the design of software more tractable. And, second, it will decrease the software maintenance costs. This second objective is, of course, a very basic one. We all know that a major and increasing part of a data processing department's budget is spent in maintaining its software system. Thus, a major portion of our software manpower is spent in maintenance rather than development of software. In fact, if present trends

to continue, one could see the day when our development effort would grind down to a
because all our software manpower would be used in maintenance of existing soft-
ware. A mathematical friend of mine proposed a nightmare scenario based on the hypothesis that this point of *zero* development is not necessarily a limiting one. He suggests that it is possible to have more than a hundred percent of the software manpower
to software maintenance, together with a negative software development effort. Astronomers would recognize this as the black hole of programming that follows the collapse of a giant software effort whose energy has burned out. Clearly, we must strive to avoid such a disastrous fate. And we must also avoid falling victim to "Conway's Law." For those of you who do not know, Conway is a Professor of Computer Science who observed that "the organization of an operating system resembles the organization
of the group that created it."

I realize that all my remarks are elaborations on the central theme that software development is lagging behind hardware development, and we must correct the situation. I assure you that I didn't come here tonight just to *Rub It In*. Software development is lagging because it is a much more difficult job. If it is any consolation to you, there is an analogy between the computer software versus hardware situation and the world picture.
you look at where the world stands today, you see that we have been extremely suc-
d in managing our *hard* discipline of production, but not so good in developing the *soft* disciplines of managing the output of our production capability.

Finally, let me explain why I selected the specific title of my talk. The *Everest* of software is not climbed without hazards. Bosses who are derisive about systems programming productivity—which typically runs 16 instructions per programmer per week—forget how often you are swept away by avalanches, how easily one gets out of breath, and what it feels like to have someone else climb up your back wearing crampons.
Harlan M , the eloquent advocate of top-down, structured programming, says the
to climb the Everest is to work smarter, not harder. You don't start at the bottom—even knowing whether there is a top; you start at the top and work your way down, the work getting easier and easier.

Thus, the moral of my talk is to improve on the old adage "you climb the mountain because it is there." Instead, you must specify, design, and implement the mountain yourselves—beginning with a specification of where you want to end up at the top. Then
can tell the rock, snow, and ice factories what you want them to build and how it
fit together. And all around the bottom of the mountain, you can attach lush valleys, bright rivers, and green forests, knowing that the people who enjoy them don't
to worry about avalanches and rock slides—or even need to know what the mountain looks like under its majestic shrouds of clouds.

KEYNOTE: SOFTWARE MANAGEMENT

J. S. Gansler
Deputy Assistant Secretary of Defense (Material Acquisition), The Pentagon, Washington, DC

The magnitude of software activity within the DoD is discussed from a cost and criticality point of view. Problem areas and their propagation through the life cycle are examined both with respect to observable manifestations and underlying causes. An overview of current Department of Defense actions to remove or diminish these problems is presented. The identified components of the solution are: (1) organizational focii with DoD and the Military Departments; (2) policy initiative; (3) practice and procedure initiatives; and (4) technology initiatives blended together around the theme of increased discipline and rigor in the software design, development, implementation, test, operation, and maintenance activities. Each of these components is discussed, and a prognosis given for ultimate success of the DoD Defense System Software Management program.

I. INTRODUCTION

Within the Department of Defense, we are presently spending over three billion dollars per year on Defense System Software (excluding Automatic Data Processing). In my opinion, we have been doing a poor job managing this increasingly important resource, and further we have been doing little research and development on the ways and means to improve it. Both of these shortcomings must change!

Today I'd like to discuss the problems, and their underlying causes which confront us in this area, to summarize the corrective actions we are now taking, and finally to solicit your help in carrying out the major new initiative we are undertaking. Neither the problems nor the proposed solutions that I will discuss are new. They have been studied at great lengths in technical meetings, and in industry for the last three years or more. The new thing that I want to convey here today is the sense, at all levels of DoD, of the need to act decisively and to act now!

Over the past few years we have had a series of studies, each of which highlighted this area of Defense System Software as one requiring change for reasons of cost and reliability. During the past year, we have put together (with much help from industry and the university communities) a preliminary plan of action, that is my topic for today.

Let me begin by saying that the main thrust of my remarks will not be directed at the traditional, general purpose Automatic Data Processing environment, the ADP community has been in existence for several decades now and there does exist an adequate organization and enough identifiable resources to attack these problems in an effective way. Instead, I will concern myself with those issues which provide the incentives for, and the barriers to good software engineering and management in the Defense System acquisition process. Recognizing, however, the need for full consistency with the ADP

Keynote address presented at the Symposium on Computer Software Engineering, Polytechnic Institute of New York, April 20–22, 1976.

community, close ties are, and must continue to be maintained to assure maximum transferability of k[illegible]ools, and techniques.

In general, [illegible] believe that because of their R&D nature, and close tie-in with other components of Defense Systems, the software practices most applicable to embedded computers are closely allied to those management practices which have been developed for hardware acquisition. I will elaborate on this later, but first let us consider the scope and the problems.

II. COST PERSPECTIVE

Software is big business within the Department of Defense. The current annual expenditure on Defense System software is now estimated in excess of three billion dollars; yet even this substantial sum is the tip of the iceberg. It includes direct costs only and represents a conservative estimate based on incomplete and nonuniform data. This uncertainty, as a matter of fact, is indicative of a clear problem in itself. The distribution of costs for a given year shows that 68% of the known costs is consumed in system development, while the remaining 32% of the known cost is classified as operation and maintenance. As we move further into the "age of computers," more systems now in development will transition to the operation/maintenance phase. This, coupled with high system longevity, may ultimately result in a five or ten to one ratio of operation/maintenance cost to development cost when viewed over the total life cycle (i.e., similar to the hardware ratio of life cycle to development costs).

III. MISSION CRITICALITY PERSPECTIVE

Over and above the cost picture, software is finding its way onto the critical path of more and more Defense Systems. Major Defense Systems currently exhibiting a critical software dependency number approximately 115, with 50% of these in the Research & Development phase and the remaining 50% in the Operations and Maintenance phase.

The functional applications of software within the DoD pervade almost every program. Software applications can be found in diverse systems from the large World-Wide Military Command and Control System and the B-1 Strategic Bomber down to miniaturized avionics packages and such ancillary operational equipment as trainers, simulators, automatic test equipment, and certain types of test ranges and test vehicles.

IV. NATURE OF THE PROBLEM

Several clearly observable manifestations such as excessive development and maintenance costs; schedule slippages and delays; excessive errors or faults; and duplication

and lack of standardization have emerged from our "lessons learned" [illegible] characteristic problems of most major Defense Systems acquisitions. These manifestations, of course, are symptomatic of underlying problems which originate far earlier in the development cycle. These include: lack of early management visibility and discipline; lack of life cycle perspective; insufficient R&D; insufficient control over expenditures; lack of standardization; lack of transferability; lack of hardware/software trade-offs; and the treatment of software as data rather than a configuration item are some of the real leverage points for relieving the cost and quality pressures underlying the observable manifestations of the problem.

Over the next few days, you will be hearing many papers which deal with these basic problems. Lets briefly explore some of the issues which are of particular relevance to this conference.

A. Inadequate Cost and Schedule Estimates

Studies by industry have concluded that there are no simple universal rules for costing software accurately, and that to estimate it accurately it is necessary to understand the nature of the individual program and the individual routine within the program. I am sure that this will remain in the situation for some time to come but we must begin now to take action to reduce the amount of "individuality" in cost estimating. In this regard, it must be said that we currently suffer from a poor historical cost data base. Not only are we unaware of what we in the DoD are spending on software in the development, production and especially operational and maintenance phases of Defense System's life; but we are unsure of the proportions of dollars which we should be spending. As it currently stands we do not allow dollars or time for the likely problems and changes which occur in each of these phases.

This situation is aggravated by the lack of common definitions, procedures, and organization in planning and managing software development. While most approaches may have merit, and some are excellent, it is not practical for Government review officials to be well acquainted with all the approaches presented to them. As a result, they cannot develop broad applicable yardsticks—they cannot really understand what is proposed or in process for each program—they cannot apply sound judgment in their management responsibilities. In general, we *believe* the estimates which are provided to us—no matter how optimistic—and budget to them. That's how the problem got started!

B. Tracking User Requirements

This is one of the biggest contributors to the high cost of software as one of your sessions recognize. The problem here again has to do with the absence of a clear understanding on the part of those managing software as to what can be accomplished with software. Frequently, they either underestimate or overestimate the state-of-the-art. At the same time it is important that software specialists be able to anticipate the likely directions of change and design software and software tools so that it is fairly easy to accommodate changes when they come.

C. People a Cost F

Each contractor has the problem of having highly trained individuals to develop and maintain the software. At a recent software conference, statistics were produced to show the turn-over time for an average programmer was about three years. There is a motivation problem for the software "production" worker, just as with the hardware "assembly line" worker. To illustrate how highly-labor-intensive software production is, it been estimated that increasing programmer productivity from an average of ten instructions per man-day to eleven could save as much as 45 million dollars per year.

D. People in Hardware/Software Trade Offs

Another part of the software environment impacting on the personnel issue is the of our ability to make the necessary trade-offs between hardware and software implementation. To assess the strengths, weaknesses, and ensuing implications of these offs on life cycle cost and reliability, we need people with in-depth understanding of both disciplines, and who can objectively perform and integrate trade-off analyses to produce a balanced system. These people are in extremely short supply and their cultivation in the future remains a major educational problem.

E. Duplication of Applications Software Efforts

I have no way of knowing exactly how much we in Defense spend on "applications" software which had already been accomplished, for some other program or programs. People with whom I have talked in the Services, tell me that it is extensive and that our efforts to control costs should begin in this area. Without clear software development standards and adequate software management in Defense Systems acquisition, the increase in costs owing to the duplication of efforts can only be expected to grow.

F. High Cost of Maintenance

A significant cost factor has been the software errors or problems discovered well after acquisition. One recent DoD study showed that Air Force avionics software costs something like 75 dollars per instruction to develop, but the maintenance of the software shown costs in the range of 4,000 dollars per instruction. The purpose of quoting these figures is not to offer them as representative numbers but to demonstrate that the of maintenance are many times those for development. I might mention here that software maintenance, in addition to correction of problems, includes updating and revision of applications programs caused by changes or expansion of the operational mission. In the future the DoD will need to take a strong look at the life cycle approach to acquiring software. This will include the formulation of design and management principles to assure software life cycle cost models, and design for ease of maintenance and program update.

G. Insufficient Software R&D

The next issue which I would like to take up concerns the need for more directed research and development on software. I refer here to the need to convert software from an art into a technology. This can only be accomplished by giving increased attention to the Research and Development of software tools. We must get away from the notion that software advancements are the sole domain of "rugged individualists" or "artists." The best practitioners, individual stars, can produce exceptionally fine software on schedule at low cost. The general run of programmers and analysts are, however, far from this standard and increase the cost, schedules, and quality of our procurements by one or two orders of magnitude. It is not very useful to say "just hire the good guys." We have no measure of either the software or its practitioners. It is all "unknown," and the fact that software is "invisible" makes it that much harder. In this regard, I am greatly encouraged by many of the efforts which I have recently observed in both the Government and business sectors to increase software production capabilities. The "software factory" concept is fast becoming an important means for effecting the necessary changes that are needed in software development practices. It involves the employment of an integrated set of tools to provide a disciplined and repeatable approach to software development and to replace ad hoc agglomerations of developmental techniques and tools with a standardized methodology. One of the key objectives of this approach has been to introduce manufacturing methods and engineering principles into those software production processes which satisfy common design, implementation, and management requirements of projects.

H. Insufficient Software Management & Control

My final area of overall concern has to do with the subject of software management in the Department of Defense, and specifically as it pertains to Defense Systems. We have become somewhat expert at knowing how to divide the responsibilities of the "requirements" and the "procurement" people in the hardware acquisition process; and there is a fairly clear line of demarcation between what is hardware and what constitutes data. Software, however, creates something of a problem, for up until recently most managers and contracting personnel were content to treat it simply as data. As costs began to soar it became obvious that some management changes were in order. If we are to manage software in the same manner as we do hardware perhaps we must begin to think of software as "property" and not solely as "data" (fully recognizing the legal implications of the term "property"). Cost data which are submitted for data items in contracts are usually only estimates and do not provide for detailed cost breakdowns for each data item. Frequently, it is difficult to get a clear and distinct separation of data costs from engineering efforts tied to a deliverable contract schedule item. We must certainly take steps to clear up this matter for software estimates.

More to the point, however, we must recognize that from a functional standpoint computer software is equivalent to hardware, and must be delivered as an active system component. This means that technical and management control is required to insure a well engineered quality product. Management instruments and disciplines influencing computer software engineering, prototyping, configuration control, quality assurance,

production control, reliability and maintainability, standardization, modular partitioning, design reviews, and life cycle costing must be applied.

V. CURRENT ACTIONS

The problems and issues I have raised are real, and they require our immediate attention. We have begun to take action aimed at improving the management situation for both the short and the long term. I would like to elaborate further on specific actions now taking place within the DoD, and to highlight the areas in which further policy guidance will soon be forthcoming.

We have created the proper organizational foci within DoD both at the OSD and Service levels. A DoD-wide software management plan which addresses all of the problems I have spoken of this morning has been derived and developed. This plan has been released and is soon to be available through the Defense Documentation Center; a DoD Directive establishing policy for the management and control, by DoD Components of computer resources and software during development, acquisition, deployment, and support of Defense Systems, has been written, co-ordinated throughout all OSD and Service organizations, and submitted to the Deputy Secretary of Defense for signature. The theme pervading all of these steps is to elevate software policy, practices, procedure, and technology from an artistic enterprise to a true engineering discipline. Or to say it another way, to treat software more like hardware throughout its complete life cycle.

VI. SOFTWARE REQUIREMENTS AND RISK ANALYSIS

The first area of emphasis under our newly formed policy initiative concerns the requirements validation and risk analysis attendent to computer resources and software. This topic will be covered in one of your sessions later today. Briefly stated, computer resource requirements with particular emphasis on software, and on hardware/software trade-offs must be reviewed, analyzed, and validated during the Concept Formulation and Program Validation phases of Defense System development, prior to the full scale development decision point. This analysis must assure conformance of planned computer resources with stated operational requirements. Risk analysis, preliminary design, hardware/software integration methodology, use of existing software modules, standardization, external interface control, security features, and life cycle system planning will be included in the review. Correctness of software, reliability, integrity, maintainability, ease of modification, and transferability will be major considerations in the initial design. The risk areas, and a plan for their resolution shall be included in the Decision Coordinating Paper at the OSD level. In addition, computer resource requirements will be continuously co-ordinated and reconciled with system operational requirements throughout system development after the decision to enter full scale development.

The effect of this policy will be to emphasize the front end technical efforts which occur prior to a major management commitment, and to insure that it is given the same attention as hardware during the early phases of system development. In addition it will

provide top level Service and OSD visibility into cost, schedule, option, and parameters at a time when subsequent development can be meaningfully impacted.

VII. COMPUTER RESOURCE LIFE CYCLE PLANNING

A computer resource plan will be developed prior to the decision to enter full scale development, and will be maintained throughout the life cycle. The purpose of the plan is to identify important Defense System computer resources acquisition and life cycle planning factors, both direct and indirect; and to establish specific guidelines to ensure that these factors are adequately considered in the acquisition planning process. Resource planning is to include equipment, software, documentation, and personnel.

This policy will place economic trade-offs, acquisition strategy, maintenance and modification decisions on a life cycle basis, and eliminate the tendency to optimize development costs, schedules, and quality at the expense of the subsequent operations and support costs. We must stop mortgaging our future in exchange for fleeting benefits during development.

VIII. CONFIGURATION MANAGEMENT OF COMPUTER RESOURCES

The next policy area concerns the configuration management of computer resources in major Defense Systems. We can no longer afford to treat software as a data element to be acquired by a one-line entry on a Contract Data Requirements List. Instead it will be treated as a full-fledged configuration item, with all the attendant disciplines and control involved. The emphasis will be on product definition, requirements traceability interface definition and control, cost and equality traceability, and the corollary control discipline.

IX. SUPPORT SOFTWARE DELIVERABLES

When it is cost-effective to do so, unique support items required to develop and maintain the delivered computer resources over the system's life cycle will be specified as deliverable, with DoD acquiring rights to their design and/or use. Examples of such support items are compilers, environmental simulators, documentation aids, test case generators and analyzers, and training aids. The provisions of the Armed Service Procurement Regulations will govern the implementation of this policy.

This policy again emphasizes the life cycle cost-effectiveness aspect of the acquisition decision. It will remove the long term dependence on a single development contractor (thereby preserving DoD's maintenance and support options for the longest possible time), and it represents a necessary, although not sufficient step toward achieving transferability of support software across mission and application lines.

X. LEST DEFINITION AND ATTAI E T CRITERIA

Specific milestones to manage the life cycle development of computer resources, including computer system and support software will be used to ensure the proper sequence of analysis, design, implementation, integration, test, documentation, operation, maintenance, and modification. These milestones will include specific criteria that measure their attainment.

This policy relates to the product definition and work accomplishment aspects of configuration management but the additional stress on quantitative demonstration criteria is significant to note. Also of particular significance is the rigorous treatment which must be accorded to test and evaluation, beginning in the earliest phases of system develop- and culminating in a complete operational test and evaluation by the ultimate military users.

XI. SOFTWARE LANGUAGE STANDARDIZATION AND CONTROL

The next policy issue deals with programming languages, which is the subject of another of your sessions this week. DoD approved High Order Programming Languages (HOLs), *will be used* to develop Defense System software, unless it is conclusively demon- that none of the approved HOLs are cost-effective over the system life cycle, and this will not be easy. Each DoD approved HOL will be assigned to a designated control agent who will be responsible for issuing the stability of the language, validating compli- of compiler implementations with the standard language specifications, gathering data as to the use of the language, for disseminating information, compilers, and tools, for improving it over time.

This policy impacts both the language selection and proliferation problems I have earlier. In general, high order languages do afford considerable life cycle benefits (particularly in the operation and support phases) even though some inefficiencies may be experienced in development. Exceptions to the use of high order languages must be ed over the life cycle, and not just for development advantages.

Our long range objective is to get down to a minimum number of DoD High Order Languages–and we are working in that direction; but our objective in achieving this standard is for cost reductions–so we must be flexible in how we apply it and how we allow it to improve over time.

XII. CO-ORDINATED RESEARCH AND DEVELOPMENT

In the area of research and development, a disciplined engineering approach to management of software design, engineering, and programming is essential as your conference indicates. We have already given funding guidance to the Military Departments to assure that this methodology is developed and is used. Specific actions underway within DoD are:

(1) Plan and execute a co-ordinated research and development program to identify and supply the technological base needed to support the policy, practice, and procedure initiatives contained in the Defense System Software Management Plan. Obviously all of the sessions to be covered in this conference are germane to this rather broad charge to the research and development community
(2) Prepare and maintain appropriate guidance documents (e.g., guidelines, checklists, handbooks, and descriptive examples) covering requirements definition, development, acquisition, operation, and support issues attendant to computer software in Defense Systems. These documents will be available for use as necessary by program managers and their staffs as well as organizations tasked with specific responsibility for developing, acquiring, operating, and supporting the computer resource elements
(3) Establish and/or maintain appropriate education, training, and experience career paths with accompanying career incentives to foster the development and retention of professional computer resource engineers, managers, and technicians.

XIII. LOOKING AHEAD

I have summarized this morning some of the more important policy actions we are now taking within the Department of Defense. In varying degrees, these techniques are being applied to all current and new Defense System programs. I think we are now getting to the point where we can impact many areas which will change the way we do business within the Defense software community. We have great need of your help in achieving these objectives. I am certain that we will be able to rely on you, as we have in the past.

MODERN SOFTWARE DESIGN TECHNIQUES

R. E. Fairley

Industrial Engineering Department,
Texas A&M University,
College Station, TX

The Software Engineering discipline is concerned with developing and utilizing systematic methodologies and techniques to design, implement, and maintain software systems. Development of systematic design techniques for software design is one of the central issues in Software Engineering.

This paper discusses the nature of the software design process, surveys some of the more popular methodologies appropriate for modern software design, reviews several notational schemes for specifying a design, and mentions the influence of the implementation language on the design process.

Design approaches discussed include top-down and bottom-up design; successive refinement; integrated top-down design, coding, and testing; programming by operation clusters; levels of abstraction; nucleus extension; information hiding; and composite design.

Notational schemes discussed include structure charts, HIPOs, pseudocode, and structured flowcharts. Programming language characteristics mentioned in the paper include the conceptual basis of a language, data types, data structures, operators, control flow mechanisms, structure of and communication between subprograms, and implementation aspects of the language.

The major theme of this paper is that design techniques and notations are available to facilitate the production of high quality software. What is now required is widespread dissemination of and experimentation with these techniques. This paper attempts to disseminate some of the existing knowledge concerning modern software design.

I. INTRODUCTION

During the past few years, increasing emphasis has been placed on the development and utilization of methodologies and techniques for designing, implementing, and maintaining software systems. The term "Software Engineering" has been coined to describe these and related activities. Design techniques for producing high quality software systems is a central issue in software engineering.

This paper discusses the nature of the software design process, surveys some design methodologies appropriate for modern software design, mentions some representational schemes for specifying a design, and describes the influence of the implementation language on the design process.

Design approaches discussed include top-down and bottom-up design; successive refinement; integrated top-down design, coding, and testing; programming by operation clusters; levels of abstraction; nucleus extension; information hiding; and composite design.

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976.

Representation discussed include str , HIPOs, pseudocode, structured flowcharts. Programming language characteristics mentioned in the paper include the conceptual basis of a language, data types, data structures, operators, control flow mechanisms, structure of and communication between subprograms, and implementation aspects of the language.

The major themes of this paper are that high quality software can be achieved only by thoughtful design and careful implementation, and that design techniques and representations are available to facilitate the production of high quality software. What is now required is widespread dissemination of, and experimentation with, these techniques. This paper consolidates some of the existing knowledge concerning modern software design.

II. THE SOFTWARE DESIGN PROCESS

The software design process is concerned with developing software systems that implement acceptable solutions to computational problems. There are two basic types of computational problems: those due to the nature of the computer itself (systems programming problems); and those generated by the user community (applications programming problems). The design techniques discussed here are applicable to the design of both systems programs and applications programs. A software development project typically starts with the formulation of functional requirements, which include a statement of the problem to be solved and the constraints that exist for its solution. The system design is derived from the functional requirements–it provides detailed descriptions of the algorithms, data structures, and interfaces that will (hopefully) satisfy the requirements. Design specifications form the basis for implementation and validation of a software system. Thus, design specifications are the link between "the what" (functional requirements) and "the how" (a software system that satisfies those requirements).

Creative aspects of the software design process include establishing a conceptual view of the system, identifying a set of high level functions to reflect that conceptualization, decoupling and decomposing the high level functions to more elementary levels, developing algorithms and data structures to implement the elementary functions, and modularizing the resulting system. Modularization involves the association of a program module with each system function, and the establishment of well defined interfaces between modules.

The design process must also result in a software system that meets certain operational requirements, satisfies various design constraints, and promotes desirable quality attributes of the system. Operational requirements are generally stated in terms of the hardware and software environment needed to support the system, execution time and memory space limitations, and the level of sophistication and number of persons required to operate the system. Design constraints are imposed by the functional requirements, and by the resources available to implement and maintain the system. The functional requirements might, for example, specify COBOL as the implementation language (perhaps in the interest of transportability), even though technical considerations would favor the use of a special purpose language more suited to the application.

Resources to implement maintain a include the hardware, supporting software, personnel (programmers, operators, and users), and time.

The particular quality attributes that are emphasized in a software system depend on the functional requirements and intended use of the system. The goal of software design is specification of systems that provide optimal levels of reliability, efficiency, flexibility, and generality for the particular class of problems that the system is designed to solve. Reliability is the most important attribute of a program or system of programs. Other quality attributes become interesting only when the system exhibits reliable behavior.

The key to software quality is design clarity. There are two aspects to design clarity: first is the clarity with which the system design reflects the functional requirements; and second is the degree to which the design specifications are mirrored in the source code implementation of those specifications. Both aspects of design clarity are essential to achieving a well designed and properly implemented software system. Design clarity contributes to all of the quality attributes, including efficiency– the performance of a well designed and properly implemented system is by definition easier to measure and tune than that of a poorly designed and badly implemented system. Similarly, a clearly designed system is easier to understand, debug, and test; thus reliability and maintainability profit from design clarity. Also, a clearly designed system will be easier to modify.

Design clarity is enhanced by the functional modularity of the system. Functional modularity is achieved by decomposing a system into distinct program modules that communicate through well-defined interfaces; each module is identified with a specific, well-defined system function. A module is a named sequence of statements that can be referred to by a collective name. Methods of implementing modules include closed subroutines, macros, and library members. The types of interfaces between modules include data, control, and service interfaces.

Functional modularity reduces system complexity and promotes design clarity by decoupling the interactions between modules. This decoupling has numerous beneficial effects: interfaces between modules are explicitly specified as part of the design process; modules can be tested either independently or in integrated fashion; errors and design deficiencies are more easily localized; and modules can be replaced or modified with minimal side effects.

The art and craft of software design consists of identifying system functions, deciding what functions to put into which modules, and establishing the interfaces between modules. However, modularization of a software system is a somewhat arbitrary process unless conceptual guidelines are followed to systematically achieve the goals of the design process. Liskov [1] discusses the fact that improper modularization may introduce additional complexity into a system in one or more of the following ways: (1) too many related but different functions in a module will tend to obscure the logic with tests to distinguish among the various functions; (2) a common function is not identified soon enough and as a result it is distributed among several different modules, obscuring the logic of each; and (3) modules may interact on common data in unexpected ways.

The effect of size and scale must be taken into account in any meaningful discussion of software design. In a small system (one written by one man in one or two months) methodical design will result in a superior product, but is perhaps not essential to the success of the project. In larger systems, the lack of a methodical approach to design will increase the complexity of the project, and may in fact render successful completion of the project impossible.

The following sections of this paper describe several design techniques that have been developed as conceptual frameworks for achieving a modular software system design. Also included are discussions of representation schemes for specifying the design, and the influence of the implementation language on the design process. The thesis of this paper is that quality must be designed into a system, and that design techniques and notational schemes are available to facilitate the production of high quality software.

III. BASIC DESIGN STRATEGIES

Two basic strategies for achieving a modular design are the "top-down" and "bottom-up" approaches. Using the top-down approach, attention is first focused on global aspects of the overall system. As the design progresses the system is decomposed into subsystems and more consideration is given to specific issues. The top-down strategy of decomposing tasks into algorithms and data into data structures has been termed "stepwise refinement," "stepwise program development," and "successive refinement" [3]. The basic principles of stepwise refinement include:

(1) Decomposing design decisions to elementary levels
(2) Isolating design aspects that are not truly interdependent
(3) Postponing decisions concerning representational details as long as possible.

Numerous examples of the stepwise program development process can be found in [3-5] One of the best known examples illustrating stepwise program development is the eight queens problem discussed by Wirth [4], and by Dijkstra [5].

The concept of backtracking is fundamental to top-down design. As design decisions are decomposed to more elementary levels it may become apparent that higher level decisions have led to an inefficient or awkward modularization of lower level functions. Thus, a higher level decision may have to be reconsidered and the system restructured accordingly. In order to minimize backtracking, many designers advocate a mixed strategy which is predominately top down, but which permits specification of the lowest level modules first. This is sometimes called the "toughest-first" design strategy.

In the bottom-up approach to software design, the designer first attempts to identify a set of primitive concepts, notions, and actions. Higher level concepts are then formulated in terms of the basic ones. System design is thus facilitated by identifying the "proper" set of primitive ideas. According to McClure [6], the bottom-up design procedure is one of concatenation, as opposed to the top-down process of decomposition. The bottom-up strategy combines features directly available in the implementation language into more sophisticated structures. These structures are in turn combined until a set of modules and data structures have been constructed from elements available in the actual programming environment.

In practice, the design of a software system is seldom (if ever) accomplished in pure top-down or pure bottom-up fashion. A predominantly top-down strategy is most successful when a well defined environment exists for software development; as for example, in writing a compiler for use with an existing operating system. When the environment is ill defined, as in the development of an operating system for a new machine, the design strategy must of necessity be a mixed strategy, or a predominately bottom-up strategy. According to Randell [7], the top-down approach is useful when constructing components to match a set of specifications, while the bottom-up approach is useful in situations where the utility of a component within the total system can be determined.

IV. INTERFACE DESIGN

The techniques by which interfaces between modules are established provide a point of reference for discussing modular design methodologies. The types of interfaces that exist between modules include: control interfaces; data interfaces; and service interfaces. Control interfaces exist in the invocations, and in the entry and exit points of the various modules. Data interfaces are established by the parameters used to pass information between modules, and by global data that is referenced in two or more modules. Service interfaces among modules are manifest in the supporting functions that modules provide for one another.

A. Control Interface Design

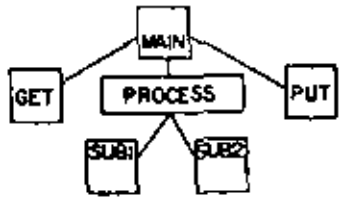
Traditional software design techniques concentrate on control interfaces. Systems designed along control interface lines are characterized by the use of flowcharts as design tools, and the program modules are typically implemented as subroutines. This strategy can produce clearly defined control interfaces, but it also tends to produce complex data interfaces.

A control interface design methodology that has yielded impressive results is the strategy of integrated top-down design, coding, and testing [8]. In integrated top-down design, coding, and testing, the design proceeds top-down from the highest level routines, whose primary function is to coordinate the sequencing of lower level routines. Lower level routines may be implementations of elementary functions (those which call no other routines), or they may invoke more primitive routines in order to accomplish their function. There is thus a hierarchial structure to a top-down system in which routines can invoke lower level routines but cannot invoke routines on the same or a higher level.

The integration of design, coding, and testing is illustrated by the following example. It is assumed that the design of the system has proceeded to the point illustrated in Figure 1. The purpose of procedure MAIN is to coordinate and sequence the GET, PROCESS, and PUT modules. These three modules can communicate only through MAIN: similarly, SUB1 and SUB2 (which support PROCESS), can communicate only through PROCESS. Some designers would allow MAIN to communicate directly with SUB1 and SUB2 while others would require that MAIN communicate with SUB1 and SUB2 by going through PROCESS.

The coding and testing strategy for the example might be as illustrated in Figure 1. The stubs referenced in Fig. 1 are dummy modules written to simulate subfunctions in support of higher level functions. As coding and testing progresses they are expanded into full functional units which may in turn require lower level functions to be

(a)



(b)

```
STRATEGY:  CODE   MAIN
           STUBS FOR GET, PROCESS, PUT
           TEST   MAIN
           CODE   GET
           TEST   MAIN, GET
           CODE   PROCESS
           STUBS FOR SUB1, SUB2
           TEST   MAIN, GET, PROCESS
           CODE   PUT
           TEST   MAIN, GET, PROCESS, PUT
           CODE   SUB1
           TEST   MAIN, GET, PROCESS, PUT, SUB1
           CODE   SUB2
           TEST   MAIN, GET, PROCESS, PUT, SUB1, SUB2
```

Fig. 1. (a) Hierarchial system. (b) Integrated top-down design, coding, and testing strategy.

them. The stub can fulfill a number of useful purposes prior to expansion into a functional unit. Stubs can provide output messages, test input parameters, deliver simulated output parameters, and simulate timing requirements and resource utilization. In this manner, a simulated system can be operational as the design progresses.

In some systems, data communication between the modules is restricted to parameter lists, while other situations allow global variables that are common to two or more modules. A reasonable compromise is to communicate data between levels via parameter lists, and to permit access to common global data by modules on the same level of hierarchy. This approach is particularly attractive when each hierarchial level is identified as a "level of abstraction" in the system design [9]. Each level of abstraction provides supporting functions for the next higher level in the hierarchy, and is supported by the level immediately below it.

The integrated top-down design strategy provides an orderly and systematic framework for system development. Design and coding are integrated because expansion of a stub will typically require creation of new stubs to support it. Test cases are developed systematically and each module is tested in a simulated operating environment. A further advantage of the integrated top-down approach is the reduction of the system integration phase of the project; the interfaces are established, coded, and tested as the design progresses. The primary disadvantage of the top-down approach is that early, high level

design decisions may have to be reconsidered when the design progresses to the lower level functions. This may require design backtracking, and considerable rewriting of code.

B. Data Interface Design

Traditional approaches to data coupling between program modules include: association of parameter lists in the calling and called modules; global variables known in two or more modules; and access to a common data base by several modules. Many transaction driven systems are designed around a large data base, which is the focal point of the design.

Liskov has described a design strategy that emphasizes data interfaces [10]. In her approach, a software system is viewed as a collection of abstract data types and operations on those data types. An abstract data type is defined in an "Operation Cluster," which defines the data type in terms of the operations that can be performed on it. For example, a stack might be defined by abstract operations such as: push; pop; return top; erase top; and empty test. The internal details of operation clusters are hidden from the modules that make use of the clusters. Thus, a stack can only be accessed by the defining operators and their parameters. This reinforces the functional modularity of the system.

A programming language called CLU is being developed to support direct implementation of software systems designed following the data interface strategy. In CLU, a cluster definition consists of four parts: (1) a description of the interface which the cluster presents to its users (cluster name, parameters, and list of operations defining the cluster type); (2) details concerning the internal representation of the data type; (3) the code required to create instances of the data type; and (4) the operator definitions. Operator definitions are similar to ordinary procedure declarations, except that they have meaning only as part of a cluster declaration.

The cluster description defines the template of an abstract data type—instances of that type are created by assigning the template name to program variables. It is therefore possible to define the abstract data type "stack" and to create and manipulate several instances of stacks in the program. The situation is analogous to the treatment of classes in SIMULA [11]. Because the manipulation of abstract data types involves their defining operations, most of the procedure calls (abstract operations) in a program are specific to, and subordinate to, the data types being manipulated in the program. In this manner, the data interfaces in the program are emphasized, and procedure calls are incidental to the manipulation of abstract data. An illuminating example of programming with abstract data types is presented in reference [10].

C. Service Interface Design

In the service interface design methodology, a software system is viewed as a set of modules that perform services for one another. Emphasis is placed on identification of set of service functions that will implement the system task. Three design strategies the service interface category are: levels of abstraction; nucleus extension; and information hiding. As originally described by Dijkstra [9], levels of abstraction was a top-down design technique in which an operating system was designed as a layered

hierarchial levels, starting with level 0 (processor allocation, real-time clock interrupts) and building [illegible] the level of processing independent user programs. Each level of abstraction is [illegible] posed of a group of related functions, some of which are external (can be invoked by functions on higher levels of abstraction), and some of which are internal to the level. Internal functions can only be invoked by other functions on the same level and are used to perform tasks common to the work being performed on that level of abstraction. Each level of abstraction performs a set of services for the functions on the next higher level of abstraction. Thus, a file manipulation system might be layered as a set of routines to manipulate fields (bit vectors on level 0), a set of routines to manipulate records (sets of fields on level 1), and a set of routines to manipulate files (sets of records of level 2). Each level of abstraction has exclusive use of certain resources (I/O devices, data) that other levels are not permitted to access. Higher level functions can invoke functions on lower levels but lower level functions cannot invoke or in any way make use of higher level functions. The latter restriction is important because lower level modules then are self-sufficient for supporting abstractions up to their level; they can be used without change as the lower level routines in other applications, or in adaptions and modifications to an existing system. In addition, the strict hierarchical ordering of routines facilitates "intellectual manageability" of a complex software system [5].

A related design technique is the nucleus extension approach described by Brinch Hansen [12]. Using this approach, the basic nucleus of a software system is identified and implemented in such a way as to permit systematic extension of the nucleus to a complete system. In the case of an operating system, the nucleus might consist of facilities to handle dynamic creation, control, and removal of processes, as well as communication between processes.

Although levels of abstraction and nucleus extension were developed specifically as bottom-up techniques for the design and implementation of operating systems, they are of much broader applicability. For instance, stepwise refinement can be characterized as a combined top-down and levels of abstraction approach to software design [1]. In this case, levels of abstraction provides the conceptual framework for placement of modules within the top-down hierarchy.

The third service interface approach to be discussed is the "information hiding" technique described by Parnas [13, 14]. Using this technique, each module is designed to hide as much information as possible from the other modules in the system. This criterion not only provides a basic design strategy, but also provides a standard for elaboration and refinement of a design. Parnas observes that the approach results in designs that are functionally modular, and that have minimal coupling between modules. This in turn provides increased clarity of the design.

An interesting aspect of the information hiding approach is the use of a non-procedural specification language to describe the functional modules. In a well known example, Parnas illustrates the conventional control interface design of a KWIC index production system, and an unconventional design of the same system using the information hiding strategy [13]. The latter design is clearly superior to the conventional design in terms of functional modularity.

In [21], Parnas discusses a major problem inherent in the implementation of all highly modular software systems that are implemented using the subroutine and macro facilities of conventional programming languages. Implementation of a processing task in numerous small, independent modules requires frequent switching between the modules. If the modules are implemented as subroutines, frequent switching will require frequent saving and restoring of subroutine linkages and environments. In a highly modular system, the overhead of subroutine linkage becomes a significa[illegible]tion of the total execution time. Conventional macros create problems when the macro definition and macro usage are ignorant of one another, which is desirable in a highly modular system.

Parnas proposes that programming languages be designed to allow groups of declarations to be defined as environments, and to allow the association of an environment with an arbitrary segment of source text. This would allow the use of macros to alter environments at compile time by the use of inline code expansion and specialized linkage mechanisms between routines. In this way, the overhead associated with altering environments at run time could be reduced.

The major disadvantage of this approach is the requirement of maintaining a mapping between the highly structured source text and the relatively unstructured object code. This mapping is needed so that the programmer can relate the object code to the source code, and so that the computing system can relate run time error messages to the source code. According to Parnas, this problem is being investigated.

V. STRUCTURED DESIGN

A software design methodology called, "Structured Design" or "Composite Design" has been described by Myers, Stevens, and Constantine [15, 16]. The conceptual approach advocated in structured design is to configure the system so that the number and complexity of connections between modules is minimized. This is accomplished by minimizing the degree of coupling between modules and by maximizing the internal cohesion of each module.

The strength of coupling between two modules is influenced by several factors, including: (1) the complexity of the interface; (2) the type of connection; and (3) the type of communication. Obvious relationships result in less complexity than obscure or inferred ones. For example, interfaces established by common control blocks, common data blocks, common overlay regions of memory, common I/O devices, and/or global variable names are more complex (more highly coupled) than interfaces established by parameter lists passed between modules.

Modules whose connections are established by referring to other modules by their functional names are more loosely coupled than are modules whose connections refer to internal elements of other modules. In the latter case, the entire content of a referenced module may have to be taken into account when updating the module that refers to it.

The type of communication between modules includes passing of data, passing elements of control (such as flags, switches, labels, and procedure names), and modification of one module's code by another. The degree of coupling is lowest for data communication, higher for control communication, and highest for modules that modify other modules.

Internal cohesion of a module is measured in terms of the strength of binding elements within the module. Binding of elements occurs on a scale of weakest to strongest in the following order:

(1) Coincidental
(2) Logical
(3) Temporal
(4) Communicational
(5) Sequential
(6) Functional.

Coincidental binding occurs when the elements within a module have no apparent relationship to one another. This result from "modularizing" an existing program into arbitrary modules, or from creating a module of unrelated instructions that appear several times in other modules.

Logical binding implies some relationship among the elements of the module; as for example, in a module that performs all input and output operations, or in a module that edits all data. A logically bound module often tends to combine several related functions in a complex and interrelated fashion; this results in passing of unnecessary parameters, and in shared and tricky code which is difficult to understand or modify. For example, a module to edit all data might be decomposed into four modules for editing master records, editing update records, editing addition records, and editing deletion records.

Modules with temporal binding tend to exhibit the same disadvantages as logically bound modules. However, they are higher on the scale of binding because all the elements are executed at one time, and no parameters or logic are required to determine which elements to execute.

The elements of a communicationally bound module are related by reference to the same set of input and/or output data. For example, "print and punch the output file" is communicationally bound. Communicational binding is higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

Sequential binding of elements occurs when the output of one element is the input for the next element. For example, "read next transaction and update master file" is sequentially bound. Sequential binding is high on the binding scale because the module structure usually bears a close resemblance to the problem structure. However, a sequentially bound module can contain several functions or part of a function since the procedural processes in a program are often distinct from the function of the program.

Functional binding is the strongest, and hence most desirable type of binding of elements in a module because all elements are related to the performance of a single function. Examples of functionally bound modules are "compute square root," "obtain random number," and "write record to output file."

According to Myers, a hierarchical tree structured design is the general solution form that will usually result in the lowest cost implementation (where cost refers to the cost of designing, coding, testing, maintaining, and modifying the system). The subdivision of functions into separate modules should be continued until each module contains no subset of elements that could be useful alone, and until each module is small enough that its entire implementation can be grasped at once. It is suggested that the implementation of a module should require between 5 and 100 executable source statements. Weinberg has suggested that a group of about 30 statements is the upper limit that can be assimilated on first reading of a module [17]. In the initial design, one should subdivide too finely when in doubt because small functions can easily be recombined later, and duplicate functions may not be identified if the subdivision is too coarse.

The concepts of "scope of control" and "scope of effect" are suggested as useful aids for determining the relative positions of modules in a hierarchical framework. The "scope of control" of a module is that module plus all modules that are subordinate to the module. In Fig. 2, the scope of control of module B is B, D, and E. The "scope of effect" of a decision is the set of all modules that contain some code whose execution is based on the outcome of that decision. Systems are simpler when the scope of effect of a decision is within the scope of control of the module containing the decision. The following example illustrates the situation.

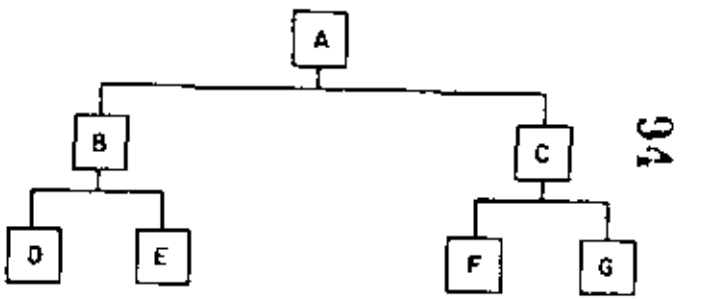


Fig. 2. Hierarchical software structure.

Referring to Fig. 2, if the execution of some code in module A is dependent on the outcome of decision X in module B then either B will have to return a flag to A, or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in duplicating some of B's function (decision X) in Module A. Duplicates of decision X result in difficulties of coordinating changes to both copies if the source code for decision X should be changed. In general, the scope of affect can be brought within the scope of control by moving the decision element upward in the hierarchical structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

VI. SOFTWARE DESIGN REPRESENTATION

In software design, as in mathematics, the representational scheme employed is of fundamental importance. Good notation can clarify the interrelationships and interactions of interest, while poor notation can complicate and interfere with good design practice. At least two levels of design specification exist: general design specifications which describe the structure of the system (what functions, what interfaces); and detailed design specifications, which describe control flow, data flow, and algorithmic considerations within the various modules. Some design representations are appropriate for stating both general and detailed specifications while some are appropriate for the other. Commonly used representations for specifying a software system [

structure char ...POs, pseudocode, and structured flowcharts. These schemes are discussed very ...ly in this paper. A comprehensive survey of design representation schemes is provided by Peters and Tripp in a companion paper to this one [22].

A. Structure Charts

Structure charts are useful during general program design as an aid in determining the functions, parameters, and interfaces of the system. A structure chart differs from a flowchart in two ways: a structure chart has no decision boxes; and the sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart.

The structure of a hierarchial system is often described using a structure chart as in Figure 3. The chart can be augmented with a module by module description of the input and output parameters. At the higher levels parameters are abstract; they become more concrete at the lower levels of the hierarchy. The lowest level routines deal with physical data objects as input and output parameters.

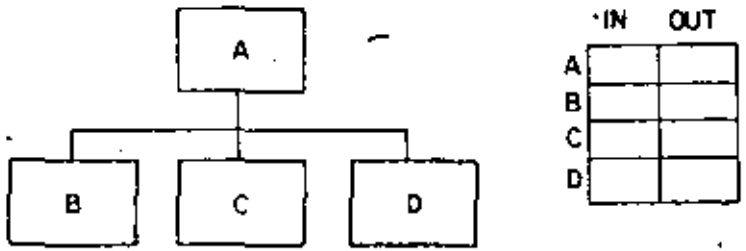


Fig. 3. Structure chart segmented with input/output table.

B. HIPOs

HIPO Diagrams (Hierarchy plus Input-Process-Output) were developed by IBM as design representation schemes for top-down software development, and as software documentation aids; design specifications and documentation are thus in the same format, which facilitates comparison of the desired product and the actual product. HIPOs are formalized structure charts; as such they emphasize functional aspects of the design rather than internal control flow mechanisms of a system.

Typically, a set of HIPO Diagrams contains a visual table of contents, overview diagrams, and detail diagrams. The visual table of contents is a directory to the set of diagrams in the package; it consists of a tree structured table of contents (called a structural overview diagram), a summary of the contents of each of the overview diagrams, and a legend of symbol definitions (see Figure 4). Overview diagrams describe inputs, a process to support the function being described, and the outputs from the process. Each overview diagram may point to several subordinate detail diagrams, the

exact number being a function of the process being described. Typical ...ts for overview diagrams and detail diagrams are presented in Figure 5.

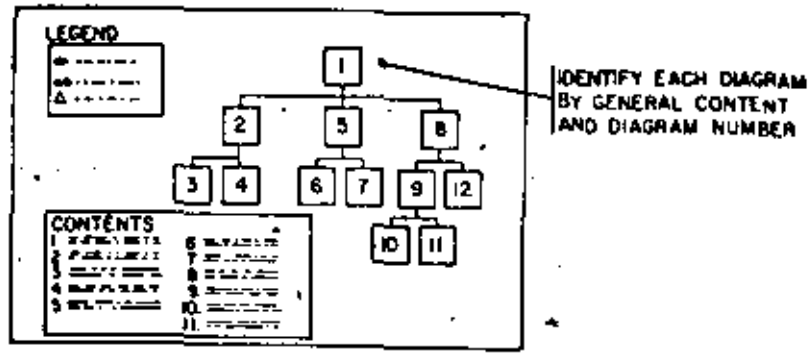


Fig. 4. Visual table of contents.

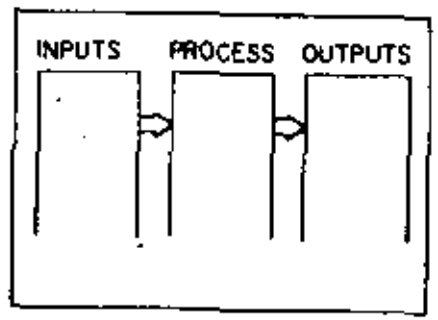


Fig. 5. Overview and detail diagram format.

C. Pseudocode

Pseudocode notation can be used in both the general and the detail design phases. The designer describes the design using short concise English language phrases that are structured by key words such as IF-THEN-ELSE, DO-WHILE, and END. Key words and indentation describe the flow of control, while the English phases describe the processing function.

As an example of a pseudocode design specification, assume that a word frequency analysis program is to be described. The program will read a set of textual records and extract each individual word from each record. A table is to be constructed that contains each unique word found in the text, and a count of the number of times each wor... occurs. When all text records have been processed, the contents of the table ar...

summary information is to be printed. The pseudocode description of the word frequency analysis program might have the following form

```
INITIALIZE THE PROGRAM
READ THE FIRST TEXT RECORD
DO WHILE THERE ARE MORE WORDS IN THE TEXT RECORD
    DO WHILE THERE ARE MORE WORDS IN THE TEXT RECORD
        EXTRACT THE NEXT TEXT WORD
        SEARCH THE WORD-TABLE FOR THE EXTRACTED WORD
        IF THE EXTRACTED WORD IS FOUND
            INCREMENT THE WORD'S OCCURRENCE COUNT
        ELSE
            INSERT THE EXTRACTED WORD INTO THE TABLE
        END IF
        INCREMENT THE WORDS-PROCESSED COUNT
    END DO AT THE END OF THE TEXT RECORD
    READ THE NEXT TEXT RECORD
END DO WHEN ALL TEXT RECORDS HAVE BEEN READ
PRINT THE TABLE AND SUMMARY INFORMATION
TERMINATE THE PROGRAM
```

In the top-down design strategy, each English phrase in the pseudocode can be expanded into a more detailed pseudocode description of that phrase. This can be continued until the design reaches the actual coding level. Pseudocode can replace flowcharts, and reduce the amount of external documentation required to describe the design.

D. Structured Flowcharts

Flowcharts are the traditional means for specifying and documenting a software system designed along control interface lines. Typically, flowcharts incorporate rectangular boxes for actions, diamond shaped boxes for decisions, directed arcs for specifying interconnections between boxes, and several other specially shaped symbols [18]. Structured flowcharts differ from traditional flowcharts in that structured flowcharts are restricted to compositions of certain basic forms. This makes the resulting flowchart the graphical equivalent of a textual pseudocode description. A typical set of basic forms and their pseudocode equivalents are illustrated in Figure 6. The basic forms are characterized by

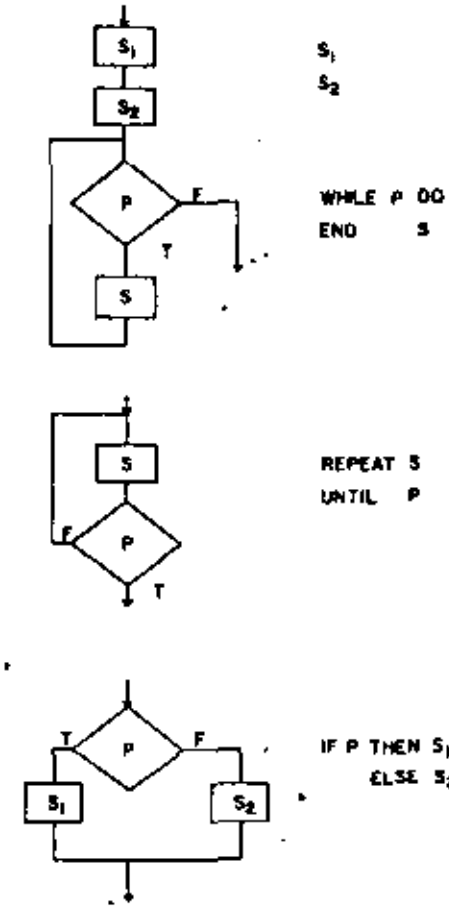


Fig. 6. Basic forms for structured flowcharts.

single entry into, and single exit from the form. Thus, forms can be nested within forms to any arbitrary level, and in any arbitrary fashion, so long as the single entry-single exit property is preserved. A composite structured flowchart and its pseudocode equivalent are illustrated in Figure 7. Because structured flowcharts are logically equivalent to pseudocode, they have the same expressive power as pseudocode. In particular, the single entry-single exit property allows hierarchical nesting of structured flowcharts to describe a top-down design, starting with general design considerations and proceeding through detailed design. Structured flowcharts tend to place increased emphasis on flow of control mechanisms due to the graphical nature of the visual image. They are thus appropriate when the decision mechanisms and sequencing of control flow are to be emphasized.

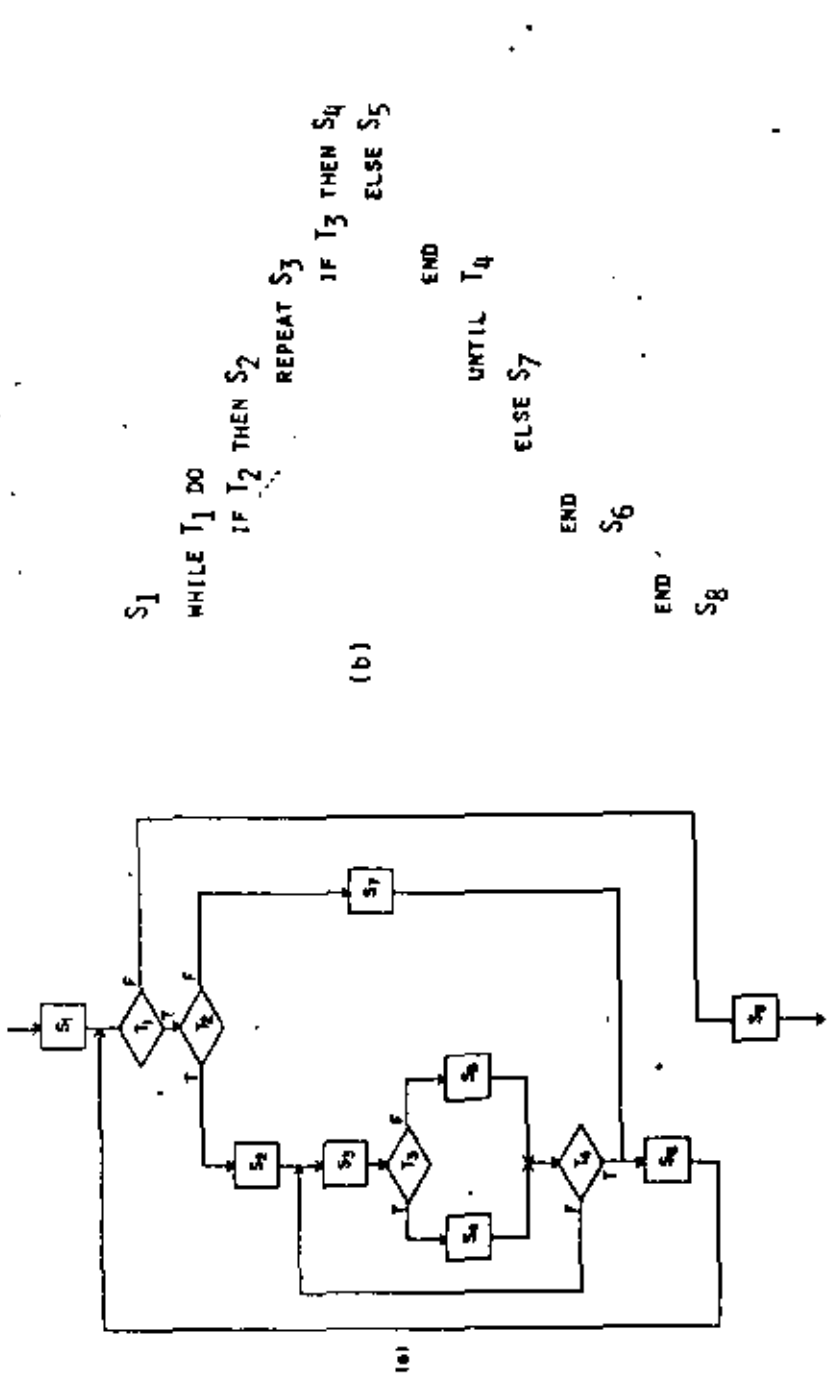


Fig. 7. (a) Structured flowchart. (b) Pseudocode equivalent.

VII. INFLUENCE OF THE IMPLEMENTATION LANGUAGE

The implementation language provides a conceptual framework and a set of basic notions for the design of a software system. Thus, a LISP-like language based on recursive function theory and list structures will encourage the designer to formulate algorithms as recursive expressions operating on lists and binary trees, while FORTRAN will influence the designer in the direction of iterative algorithms operating on arrays. Implementation of a software system is simplified when the data types, data structures, algorithms, and interfaces described in the design specification correspond to notions supported by the implementation language.

The conceptual aspects of a programming language are manifest in the data types, operators, and control structures of the language. Data types include basic and structured data types. The basic data types available in a programming language are typically a subset of the data types supported by hardware, and may include any or all of: integers; floating point numbers; decimal numbers; characters; logical values; address pointers; and bit vectors. Mechanisms for structuring basic data types include arrays, hierarchical structures, character strings, lists, trees, graphs, sequences, and sets. The ease of writing algorithms to transform and manipulate data of a given type (basic or structured) is determined by the operators provided for that data type. FORTRAN and ALGOL 60 provide arithmetic, relational, and logical operators for the basic data types of integer, floating point, and logical. A subroutine capability allows the user to implement abstract operators on basic and structured data. Newer language, such as PL/1 and APL, provide some built-in operators for the structured data types, as well as a subroutine capability that permits user defined abstract operations.

The control structures in many programming languages reflect the basic architecture of sequential machines; in the absence of an explicit control construct execution proceeds from one statement to the next in lexicographic order. Explicit control statements include various forms of conditional statements, iteration mechanisms, and subprogram calls and returns. Conditional statements include the arithmetic and logical IF of FORTRAN, the nested IF-THEN-ELSE statements in ALGOL 60, success and failure exits in SNOBOL statements, and the COND expression of LISP. Iteration mechanisms include looping statements (such as the FORTRAN and PL/1 DO statement, and the ALGOL 60 and PASCAL FOR statement) in which initialization, incrementing, and exit testing are described in the statement forms, as well as loops constructed using IF statements and GOTOs in which initialization, incrementing, and testing are handled explicitly in the source text.

The current interest in structured programming is motivated by the desire to provide control constructs that preserve the clarity of the design specifications in the source code implementation of the design. Basic premises underlying the use of structured control mechanisms are: (1) each basic construct must preserve the single-entry, single-exit property; and (2) no basic construct permits backward transfers of control in the source text (except the implicit transfers in looping constructs). Single-entry/single-exit control structures can be hierarchically nested within other control structures to any arbitrary depth, and indentation of nesting levels can facilitate readability of the source code. When the assumptions of nested single-entry/single-exit constructs and no backward transfers are satisfied, the dynamic execution flow through the program can be made to correspond closely to the static structure of the source text. The source text is thus highly readable and makes a significant contribution to its own documentation.

The use GOTO statements has been criticized by structured programming advoca because the GOTO provides an extra mechanism for structuring control flow, and it is quite easy to (intentionally or unintentionally) violate the basic premises of structured programming. D'Ison lists the following positive benefits to programming without GOTOs [19]:

(1) The programmer is forced to discipline his thought processes, and to formulate his logic according to an appropriate structure
(2) The programmer finds himself looking for similarities rather than differences in subportions of his problem. Rather than simply generating conditional branches to many program locations to handle a number of cases, he is encouraged to handle them together, perhaps with additional use of variables serving as parameters to differentiate the cases
(3) The reader or inheritor of a program has a much easier time following program logic if he can read the program sequentially rather than with constant page flipping through the program listing.

Much current research in structured programming is concerned with the development of control structures that allow the design to be clearly implemented in the source code notation without unduly restricting the programmer or forcing him into unnatural modes of expression.

Control sequencing of subprograms in a given programming language has a strong impact on the resulting modularity of a software system implemented in that language. Subprograms can be internal or external, they can have simple call-return structures, or multiple entry, multiple return structures, they can be recursive, interrupt driven, co-routines, attached as parallel subtasks, or scheduled for later execution [20]. Data may be communicated in parameter lists, through common memory regions, or through global variables. Parameter lists may pass parameters by reference, by value, or by name. The types of parameters that can be passed may be restricted to numerical valued variables, or may include procedure names, labels, and pointers.

In addition to selecting a language appropriate to the application, numerous implementation details will influence the quality of the software produced. The implementation features of interest may include:

(1) Compile time and/or execution time efficiency
(2) Memory space utilization
(3) Adequacy of error messages
(4) Debugging options
(5) Adherence to formal standards
(6) Stability of the implementation
(7) Extent and quality of documentation.

The stability of an implementation is revealed by answers to a series of questions such as: Who maintains the implementation? At what level of support? When was the last update released? How long has the language been installed? How often is it used? What are the experiences of other users?

Documentation should be clear, concise, and well-structured. The documentation should include an introductory users manual, an authoritative reference manual, and explanations of mac dependent implements aret. All documents sh be cross-referenced and indexed to permit rapid access to any desired level of detail. Implementation details of ambiguous, incomplete, and inconsistent language features should be noted, as well as extensions to, and sub-setting of the language. Finally, the documentation must be consistent with the actual implementation of the language.

VIII. SUMMARY

This paper has discussed various aspects of the software design process, including methodical approaches to software design, notational schemes for describing the design, and the influence of the programming language on the design process.

Design approaches discussed included top-down and bottom-up design; successive refinement; integrated top-down design, coding, and testing; programming by operation clusters; levels of abstraction; nucleus extension; information hiding; and structured design.

Representation schemes discussed included structure charts, HIPOs, pseudocode, and structured flowcharts. Various language concepts and their influence on the software design process were mentioned.

The themes of this paper have been that high quality software can only be achieved by thoughtful design and implementation, and that design methodologies and notations do exist to facilitate the production of high quality software. All of the techniques described have strengths and weaknesses which make them more or less appropriate in particular circumstances, and no single technique is clearly superior to all others in all situations.

Current and future directions in software design methodology include:

(1) Critical assessment of the various design techniques
(2) Development of design notations and programming languages for specifying and implementing the design
(3) Development of methods for integrating the new design and implementation techniques into existing technology
(4) Development of pedagogical techniques for teaching modern software design and development
(5) Determination of the proper role for computer aided and automated systems in software specification, design, documentation, code generation, testing, and maintenance
(6) Overcoming the resistance of software development organizations to the utilization of new methodology and techniques.

ACKNOWLEDGEMENTS

This work supported in part by NSF Grant DCR 74-24546 and by NA[illegible] NAS 5-20715.

REFERENCES

[1] B. H. Liskov, "A Design Methodology for Reliable Software Systems," *FJCC Proceedings* (1972).

[2] N. Wirth, *Systematic Programming: An Introduction*, Ch. 15 (New Jersey: Prentice-Hall, 1973).

[3] ——, *Systematic Programming: An Introduction* (New Jersey: Prentice-Hall, 1973).

[4] ——, "Program Development by Stepwise Refinement," *Comm. ACM*, 14, No. 4 (April 1971).

[5] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* (New York: Academic Press, 1972).

[6] C. L. McClure, "Top-Down, Bottom-Up and Structured Programming," *Proceedings of the First National Conference on Software Engineering*, DC (September 1975).

[7] B. Randell, "Towards a Methodology of Computing System Design," *1968 NATO Conference on Software Engineering*.

[8] C. L. McGowan, and J. R. Kelly, "Top-Down Structured Programming Techniques" (New York: Petrocelli/Charter, 1975).

[9] E. W. Dijkstra, "The Structure of the "THE" Multiprogramming System," *Comm. ACM*, 11, No. 5 (1968).

[10] B. Liskov and S. Zilles, "Programming With Abstract Data Types," *ACM SIGPLAN Notices*, 9, No. 4 (1974).

[11] J. Palme, "SIMULA as a Tool for Extensible Program Products," *ACM SIGPLAN Notices*, 9, No. 2 (1974).

[12] P. Brinch Hansen, "The Nucleus of a Multiprogramming System," *Comm. ACM*, 17, No. 4 (1974).

[13] D. L. Parnas, "A Technique for Software Module Specification With Examples," *Comm. ACM*, 15, No. 5 (1972).

[14] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules," *Comm. ACM*, 15, No. 12 (1972).

[15] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Syst. J.*, No. 2 (1974).

[16] G. J. Myers, "Reliable Software Through Composite Design" (New York: Petrocelli/Charter, 1975).

[17] G. M. Weinberg, *PL/1 Programming: A Manual of Style* (New York: McGraw-Hill, 1970).

[18] N. Chapin, "Flowcharting with ANSI Standard: A Tutorial," *ACM Computing Surveys*, 2, No. 2 (1970).

[19] M. Elson, "Concepts of Programming Languages," Science Research Associates (1973).

[20] T. W. Pratt, *Programming Languages: Design and Implementation* (New Jersey: Prentice-Hall, Inc., 1975).

[21] D. L. Parnas, "On the Need for Fewer Restrictions in Changing Compile-time Environments," *ACM SIGPLAN Notices*, 10, No. 5 (1975).

[22] L. J. Peters and L. L. Tripp, "Software Design Representation Schemes," *Proceedings of the Symposium on Computer Software Engineering* (New York: Polytechnic Press, 1976).

SPECIFICATION VERIFICATION–A KEY TO IMPROVING SOFTWARE RELIABILITY

P. C. Belford and D. S. Taylor
Computer Sciences Corp., Huntsville, AL

The problem of providing valid, reliable software is one of the major limiting factors in using computers to solve complex technical problems. In their rush to develop better reliability assessment techniques and/or better designed-in reliability methodologies, many software analysts fail to discern one of the major stumbling blocks that frequently negates any attempt to assess and, therefore, improve the software reliability. That is, no new programming philosophy will ever improve software reliability if the system specifications upon which it is based are incorrect or have been incorrectly translated. This paper describes the importance of verifying systems specifications before any system design and proposes a technique for accomplishing this objective.

The implementation of a comprehensive verification process that accepts system specifications as currently written requires the development of a top-down method of unambiguous translation of system requirements. In addition, this verification process must support configuration accountability and ensure total system traceability at all levels. A multilevel automata structure satisfying all of these requirements is suggested in this paper. A model is presented that: (1) describes a total system in general terms at the system specification level; (2) traces system requirements through the subsystem level, and (3) delineates the detailed system. System functional and performance requirements, as described in this model, are totally traceable and can be verified at each level of translation.

Finally, a verification simulator is suggested which accepts the translated system specifications as input and verifies performance requirements. This simulator is capable of pointing out any inconsistent and incomplete requirements.

I. INTRODUCTION

The problem of producing reliable, maintainable software is one of the major limiting aspects of the application of computers to industrial and scientific processes; this is particularly true with reference to complex technical problems [1-5]. In the rush to develop better reliability assessment techniques and alternative designed-in reliability philosophies, the concomitant software methodologies have failed to address the one major stumbling block which most often negates any attempt to improve the software reliability: no programming philosophy will improve software reliability if the underlying system specifications are erroneous or have been incorrectly translated. The purpose of this paper is: (1) to illustrate how software quality could be improved by verifying the supporting specifications; and (2) to suggest an approach to achieving this objective.

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976

'EED FOR VERIFYING THE SPECIFICATION

The importance of having completely verified specifications prior to any actual software system development has long been recognized as a major time-saving step in the evolution of any large computer-based system. Current vague definitions and measures of software reliability are due to the amorphous nature of these specifications, as noted by numerous software reliability experts [6-10]. These software experts believe that definitive specifications are fundamental to the development of reliable software.

A secondary requirement for specification verification stems from the fact that the specifications document itself represents the only real interface between client and developer. Without a verified baseline from which to work, there is very little that can be done to insure a common understanding of the system between those who will use it and those who developed it.

Since the specifications document represents an official interpretation of the overall system requirements, the efforts needed to obtain internal clarity and cohesion in the specifications statement provide the user a final chance to insure that system performance will be forthcoming prior to acceptance testing. Because of this, the first step in any software development process must be a verification of the system specifications [11]. This verification process must trace system requirements from one level of specification to another and also trace each system requirement within the specification document. The verification process is intended to point out inherent conflicts, discrepancies, omissions, and to insure the completeness of each requirement defined in the system specification.

A large number of man-hours can be wasted designing, coding, and testing software which was developed from erroneous specifications. A typical large-scale software development cycle illustrating this point is represented in Figure 1. This figure indicates how system specification errors which are not discovered until late in the development cycle can affect already developed code. Errors in system specifications of this type will necessitate design and recoding of already validated software and will also invalidate any tests which were previously accomplished [7, 8]. Because of the complexity of the interdependent relationships in this situation, system verification after code generation all too often becomes merely verification of the generated code; this results in total system reliability being completely dependent upon, or a function of, the validation process.

Figure 2 illustrates a preferred software development cycle over that shown in Figure 1. While this approach is not new, it has been widely implemented in the software industry. The major reason that this approach has not been followed is that verification techniques have historically been unstructured, manual procedures. Verification techniques of this unstructured, manual type require so many personnel (who individually know and understand several different sections of the specifications) that these techniques are physically and economically unrealistic. Additionally, the verification process must also include a functional simulation to demonstrate system capabilities to meet timing, accuracy, and overall performance requirements. The functional simulation produced by the verifier must be capable of testing the validity of system requirements, and this should be done independently of the computer upon which these requirements will be implemented.

A verified specification becomes a baseline upon which the client, the developer, and even the ultimate user can each confidently place his trust. In so doing, the client

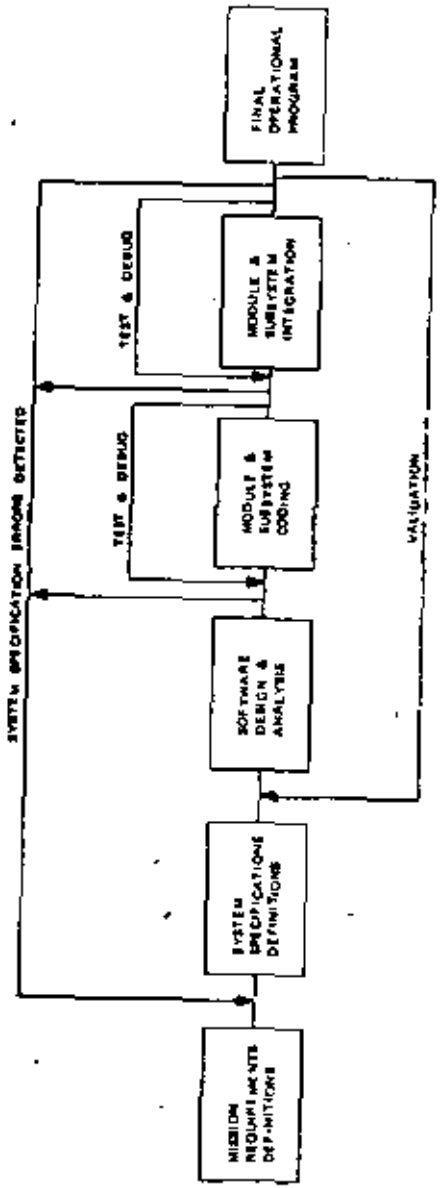


Fig. 1. Typical large-scale software development cycle.

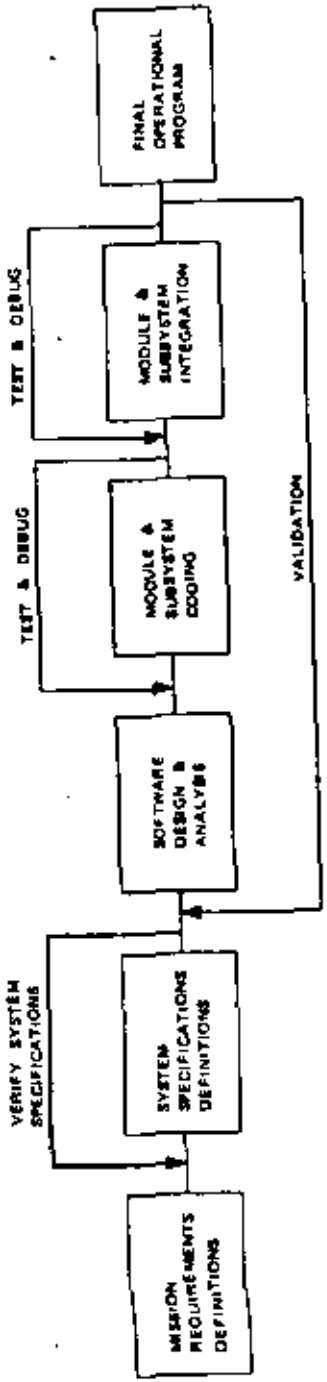


Fig. 2. Preferred large scale software development cycle.

would have a firm foundation upon which to base his software acceptance; the develc would have confidence that the development process would proceed based on sound specifications that are not contradictory or ambiguous; and the last man in the chain, the user, would not only understand what he is getting, but he would have a strong baseline for a maintainable system.

It is apparent that inadequate system specifications are the weakest link in the software development cycle and that verification of this basic element of the system is essential [11]. It would, however, be completely inappropriate for the software co munity to attain a capability to develop error-free software if the additional costs by applying such techniques are prohibitive. The overriding purpose of this paper, th fore, is to present a cost effective verification process which can be applied to system specifications.

III. THE SPECIFICATION VERIFICATION PROCESS

Any effective verification technique must be both efficient and conclusive, which requires both a coordinated manual and automated effort [12 - 15]. Formal languag and directed graph theory are mathematical systems that support the modeling of real systems so that such systems can be more discretely defined [16]. These modeling te niques best lend themselves to an automated verification process [17 - 19]. By decon posing a specification so that it represents an unambiguous grammatical form and by representing or defining this structure as a form of a directed graph, the verification process is reduced to an automated process that can be described in a computer progr This automated process will be referred to in this paper as a verifier.

There are three tasks which must be defined before a verifier can be developed. F the specification grammar must be developed; second, a structure must be defined for this grammar; and finally, a functional simulation must be generated from the structu

When developing a specification grammar, certain properties must be satisfied in order to provide continuity and form to the language. The language thus formed is a set of all the strings of terminal symbols wherein a sentence is a sentential form (a str of symbols derivable from the grammar) of the grammar consisting of only terminal sy bols. Therefore, a sentence is a grammatically valid string of terminal symbols describe in the language. However, the single sentences which make up specifications, taken as individual productions of a grammar, do not always describe any requirements or f tions that can be fully understood or comprehended at first reading. Rather, it is the groups of sentences usually organized into paragraphs, that describe requirements whic can be comprehended as action areas to be translated into software requirements. The fore, it makes sense to describe the grammar in terms of the requirements instead of th sentence. Specification paragraphs typically relate one-to-one to a functional require ment. A typical specification paragraph might describe the general concepts of a requi algorithm in its first sentence. The second sentence might define the types of input parameters that must be accepted by that algorithm. Finally, the last sentence might describe the response expected in a given environmental instance and any other pertinent external constraints impacted on the algorithm.

The production of an individual sentence such as that described in the paragra above would probably be unclear by itself, and therefore require special rules on

collection : ouping of these sentences into a paragraph in order to describe a requirement. The primary objective of the specification grammar is to redefine the system specifications in requirements terms that are technically unambiguous. This means that if the grammar is defined in terms of specification paragraphs instead of sentences, then the productions of the grammar will define one requirement for each functional response or operation. This eliminates the need for any correlation between sentential forms and simplifies the mathematical model and the symbolic logic.

Since the grammar of the language is now defined in terms of requirements instead of in disjointed sentences, the grammar interlanguage description becomes easier to define but more difficult to produce. In other words, if one accepts a conclusion that there is no predefined mathematical approach to the development of a production routine, then only the translator has the function of describing the requirements in terms of the grammar interlanguage. This, in turn, strongly implies that an automated translator is impossible to achieve.

It thus becomes intuitively obvious that completely automated translators for specification verifiers are not the answer to today's software reliability problem. If one could entirely automate the verification process, it would require that the specification be written in an unambiguously defined grammar of some previously defined and accepted language. Implementation of the ideas expressed in this paper would not require that specifications be written in any manner other than that in which they have been written in the past. This would require that a new methodology be defined and adhered to by the specification writers. Today's problems are complex enough for these writers without requiring them to complicate their current process by decomposing and documenting requirements in an unambiguous manner. We do believe, however, that it is desirable to show that specifications, typically written in an ambiguously defined grammar following some structural guidelines such as MIL-STD-490, can be manually transformed from this accepted media.

We also believe that this transformation can be done by using predefined grammatical productions which can be made into unambiguously defined structures and then be automatically verified and simulated.

IV. THE SPECIFICATION VERIFICATION GRAMMER

The specification verification grammar described in this paper is not intended to be as structurally defined as say the Backus-Naur-Form (BNF) notation of ALGOL 60 [20], but rather it is a structured description of the transformation process required to convert the specification into a directed graph. The syntactic elements are described below as a production of a specification paragraph.

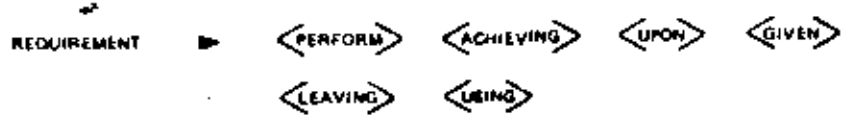


Fig. 3. Syntactic elements

These syntactic elements will not produce terminal symbols in grammar, but rather will describe the requirement in terms of Figure 4.

PERFORM	⊫	THE PROCESS (FUNCTIONAL REQUIREMENT)
ACHIEVING	⊫	THE RESPONSE (OUTPUT)
UPON	⊫	A SET OF STIMULI (INPUT)
GIVEN	⊫	A SET OF INITIAL CONDITIONS AND CONSTRAINTS (LOGICAL CONNECTIVITY)
LEAVING	⊫	THE NEW STATE OF THE SYSTEM (PERFORMANCE REQUIREMENT)
USING	⊫	THE ELEMENTS OF THE SYSTEM (DESIGN CONSTRAINTS)

Fig. 4. Requirement description.

Since a fundamental property of ambiguity is that it is not axiomatic, i.e., there exists no algorithm which will determine the unambiguity of any grammar, then in order to eliminate ambiguity, it is necessary to manually translate the specification paragraph into a grammar such as that described in this paper. This fact implies that the manual transformation process is in itself a mini-verification process since any ambiguities which cannot be resolved must be verified and resolved prior to their translation into the grammar proposed herein. The grammar of this verification language can be represented by the following model. (It is this model which is used as input to the verifier).

Fig. 5. Functional model.

In order to provide compatibility with the verification technology, and at the same time to permit computer processing and simulation of the requirements, the specification language is form oriented and only one input form is utilized. This input form is called Decomposition Element (DE). The Decomposition Element when completed contains:

(1) Functional requirement
(2) Stimulus
(3) Response
(4) Performance requirement
(5) Subsystem responsibility
(6) Specification paragraph references
(7) Design constraints
(8) Logical connectivity.

There is one Decomposition Element for each functional requirement at

in the specification. The DE is generated manually and loaded into the verifier for verification. This verification process syntactically analyzes each DE for validity; connects the DEs together into a directed graph to link common functions; semantically analyzes the connected DEs; and determines the consistency, closure, traceability, and completeness of the generated directed graph.

The basic problem is defining the terminal symbols. These terminal symbols are a function of the problem or class of problems which the specification is intended to solve. Ballistic Missile Defense specifications utilize a different set of terminal symbols in contrast to those used in Air Traffic Control specifications. The set of terminal symbols required for any problem can be flexibly defined by the verifier and thereafter fixed. Consequently, a unique set of verifier input terminal symbols for each problem or class of problems is thereafter used to define and correlate the individual node of the directed graph.

However, the ambiguity in an English language specification is sometimes a function of the reader's interpretation. In transforming the specification to unambiguously described Decomposition Elements, the reader may resolve the ambiguity by terminology in context, or he may simply fail to perceive and apprehend the ambiguity. Ambiguities not resolved by the verifier will be discovered by the automatic verification process. Many unresolved ambiguities will thus be identified as incomplete, inconsistent, or redundant functional requirements.

An example of the decomposition process for the "Detect and Verify Objects" requirement in the U.S. Army's Site Defense System requirements specification is in the functional description for "Schedule Search Pulse" subfunction:

> "This function includes the scheduling of the transmission and reception of all search/verify pulses and the preparation of all data needed by the radar to process the return. It begins with the receipt of a request to schedule from the Search Raster Generation function or the Process Search Return function and ends when the transmission message is placed at the interface to the radar."

Other pertinent information which was available from the same specification included that the function was to be allocated to the Data Processing Subsystem and that it must have a minimum impact on the radar timeline.

The Decomposition Element generated from these requirements is shown in Figure 6.

PERFORM	▶	SCHEDULE_SEARCH_PULSE
ACHIEVING	▶	RADAR_MESSAGE
UPON	▶	PULSE_SCHEDULING
GIVEN	▶	GENERATE_SEARCH_RASTER OR PROCESS_SEARCH_RETURN
LEAVING	▶	MINIMUM_TIMELINE_IMPACT
USING	▶	SCHEDULE_TRANSMISSION AND SCHEDULE_RECEPTION

Fig. 6. Schedule search pulse requirement.

This data is now ready to be loaded into the verifier for syntactic analysis and will be utilized in generating the directed graph of the "Detect and Verify Objects" function. A directed graph, digraph, is a network of points called nodes connected by dilinks. A dilink is a connected ordered pair of distinct nodes. Dilinks are further delineated in this approach in terms of the verifier into either local or global dilinks, where local dilinks connect nodes within the digraph and global dilinks only connect to one dilink in the digraph [21]. Nodes are either functional or logical depending on the singularity of the dilinks terminating and originating at the node.

The digraph is then utilized by the verifier to ensure consistency within the graph and also to ensure traceability, closure, and completeness between various levels of digraphs associated with various levels of functional requirements.

V. AUTOMATIC VERIFIER

The entire verification process can now be automated using a verifier to break down the input components which are compared to each other and to previous specification components. This verifier is an automated process which is subdivided into two basic components - analyzer and synthesizer, to support verification and simulation according

The analyzer, which is completely dependent upon the Decomposition Elements, has the function of scanning each element and decomposing it into its syntactic entities (e.g. lexical analysis) and also syntactically analyzing these entities for validity. Once these DEs are syntactically validated for a given level of functional requirements, they are organized into a digraph through the use of the common stimulus/response dilinks and the connectivity of the logic nodes [22].

For example, assume that the six Decomposition Elements presented in Fig. 7 were translated from the Detect and Verify Objects function previously described. Through the use of the Boolean connection matrix shown in Fig. 8, the connectivity of stimuli and responses can be organized into the directed graph shown in Figure 9. A unit element in the connection matrix represents a path of functional connectivity.

Dilinks are represented in the digraph by directional lines where the global stimulus dilink has only a successor node and the global response dilink has only a predecessor node. All other dilinks are local and have both predecessor and successor nodes. Nodes in the digraph are either presented as functional (rectangular) or logical (circular).

The digraph shown in Fig. 8 is seen to be connected with one global stimulus, Search Sector Allocation, and one global response, Verified Detection. To ensure traceability and completeness within the functional requirements of the specification there should be a functional node in the next higher level requirements digraph, probably called "Detect and Verify Objects," which has a stimulus dilink and a response dilink matching the global dilinks of the digraph shown in Figure 8.

The analyzer component of the verifier provides the capability of verifying system requirements, ensuring a one-to-one representation between the simulation and the specification, and building a requirements data base capable of providing visibility and traceability into the design process.

STIMULUS	FUNCTIONAL ANOMALY	RESPONSE
SEARCH SECTOR ALLOCATION	GENERATE SEARCH RASTER	PULSE SCHEDULING
PULSE SCHEDULING	SCHEDULE SEARCH PULSE	RADAR OPERATIONS DATA
RADAR OPERATIONS DATA	PROCESS SEARCH PULSE	RADAR SEARCH RETURN MESSAGE
RADAR SEARCH RETURN MESSAGE	PROCESS VERIFY PULSE	VERIFY PULSE RETURN
RADAR SEARCH RETURN MESSAGE	PROCESS SEARCH RETURN	PULSE SCHEDULING
VERIFY PULSE RETURN	PROCESS VERIFY RETURN	VERIFIED DETECTIONS

Fig. 7. DEs for Detect and Verify Objects function.

The verification process involves such checks as:

(1) Consistency—Each node in the digraph must be either functional or logical; each dilink in the digraph must be either local or global; and the set of all nodes and dilinks in the digraph must be connected.
(2) Traceability—The global stimulus dilink(s) and global response dilink(s) in sublevel digraphs must map one-to-one in quantity, direction, and value to simply connected dilinks in the next higher level digraph

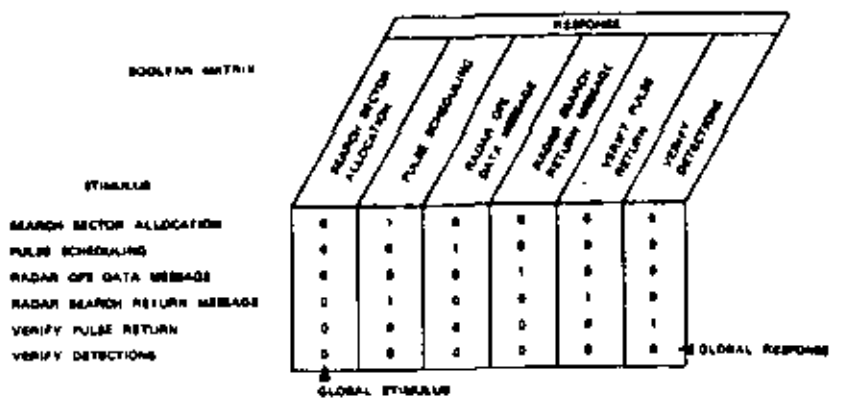


Fig. 8. Connection matrix for the Detect and Verify Objects function.

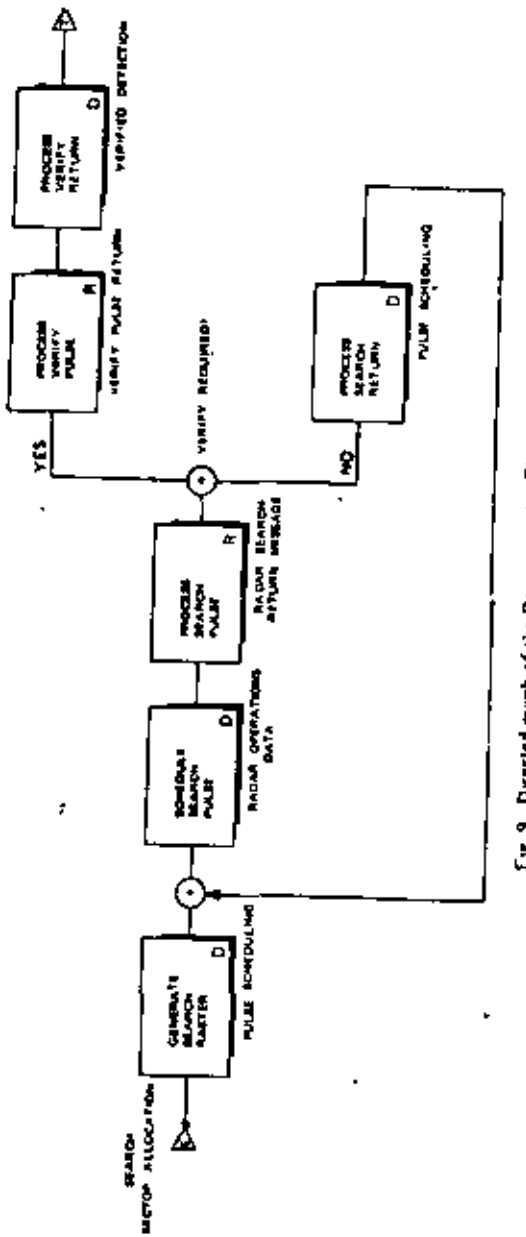


Fig. 9. Directed graph of the Decomposition Element.

(3) Completeness—All sublevel digraphs are traceable to functional nodes in the next higher digraph these functional nodes exhaust the set at that level.

The synthesizer, on the other hand, is oriented toward requirements simulation. That is, the analyzer will pass to the synthesizer all levels of the specification broken down into internal form. This internal form is then capable of being modeled into a functional simulation.

After all levels of requirements in the specification have been verified by the analyzer, the synthesizer enables each requirement, DE, at the lowest level to be modeled. These models can be individually validated in terms of the performance requirements on each Decomposition Element.

By using the Decomposition Elements to generate the simulation models, assuring a one-to-one relationship between the models and the requirements; and by simulating the system against the levels of performance requirements in a bottom-up fashion, one level at a time, the same simulator could be utilized to verify the various levels of requirements. For example, since a one-to-one relationship exists at each level between functional and performance requirements, the lowest level functional models would be measured against the lowest level performance requirements. These functional models could then be grouped together, by directed graphs, and simulated against the next higher level of performance requirements, until the overall system requirement is eventually assured. The basic approach is that of top-down verification and bottom-up simulation.

VI. CONCLUSIONS

It might be concluded that this verification process could be taken one step further, i.e., that the ideal situation would be to completely simulate the specifications directly from their documentation form. For this to be accomplished would require that specifications be originally documented in an unambiguous manner and this requirement would place an additional burden on the author of the technical specifications. However, assuming that this requirement could be met, the author of the technical specifications would have now stated the requirements in a language that would permit a computer to read and analyze the document. This makes it logical to conclude that if the specifications were unambiguously defined and if there existed a compiler that could accept this source as input, then the code generation process could be automated (assuming an elaborate verification process similar to the one described in this paper were to be used). In addition, if the code generation process could be automated, it would eliminate the need for a validation process, because after the compiler itself was validated, there would be no need to validate the object (output) code. The object code would always reflect the requirements described by the input.

However, this ideal solution is an "overkill" technique and is not considered practical for most of today's systems. The design and development of such a special compiler and verifier would have to be strongly application-oriented and would involve too much development time for state-of-the-art advanced software, not to mention the time and

investment in the verification generation process. In fact, if such a special compiler and verifier were to be used, it would not be over-critical to expect that (since the software development effort would be automated) all design, management, and implementation concepts normally associated with the development cycle would now have to go into the specification generation process.

The optimum solution should be one which uses the specifications in their current form and attempts to unambiguously redefine them in terms that can be used by the verification and software development process. The basic top-down verification concept described in this paper is intended to yield precisely this type of environment, i.e., a verification tool, a software development tool (in terms of a structured sequential requirements definition) and a configuration management tool. Once the specifications are verified, changes can be easily re-verified and impact assessment can be effectively determined using this system.

Another key term that must be applied to reliability (and reliability is directly proportional to verification, validation, and development process), is maintainability. The directed graphs described in this paper form the basis for future maintenance systems, including change management and training.

This paper has described a technique for the effective verification of specifications. As previously discussed, this verification is a key discipline required in the development of reliable software. By verifying the specifications, and thereby verifying their inherent requirements, the software developer need only be concerned with the validation of the generated code. It follows that this situation underscores and emphasizes that the responsibility for reliability in the system development process lies with both the software developer and the specification originator.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the assistance and support in this project of Dr. Carl G. Davis of the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC).

This work was supported by BMDATC under the Data Processing System Requirements Project, contract DASG60-75-C-0125.

REFERENCES

[1] C. V. Ramamoorthy et al., "Reliability and Integrity of Large Computer Programs," University of California, Memo No. ERL-M430 (March 1974).

[2] B. H. Liskov, "Guidelines for the Design and Implementation of Reliable Software Systems," The MITRE Corporation, MTR 2345 (1972).

[3] D. I. Good, "Developing Correct Software," *First Texas Symposium on Computer Systems*, Austin, TX (June 1972).

[4] H. Sackman, *Computers, System Science, and Evolving Society* (New York: John Wiley & Sons, Inc., 1967).

[5] T. B. Steel, Jr., "The Development of Very Large Programs," *Proceedings of the [illegible] DC*, I, 231–35 (1965).

[6] R. L. [illegible] Jr. and C. V. Ramamoorthy, "A Study in Software Reliability and Evaluation," Univers[illegible] Texas, TM No. 39 (February 1973).

[7] C. R. Vick, "Specifications for Reliable Software," *Proceedings of EASCON 74*, DC (October 7-9, 1974).

[8] "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's," United States Air Force, CCIP-85, SAMSO, XRS-73-1 (February 1973).

[9] B. Elspas, M. W. Green, and K. N. Levitt, "Software Reliability," *Computer* (January/February 1971).

[10] H. Liebowitz, "The Technical Specification - Key to Management Control of Computer Programming," *AFIPS Conference Proceedings*, DC, 30, 51-59 (1967).

[11] A. E. Nashman, "Software Development Management: The Key to Quality Software Products," *IEEE Electronic and Aerospace Systems Conference* (1974).

[12] R. W. Wolverton, "Software Reliability Modeling, Prediction, and Measurement Methodology," TRW Systems Group (July 15, 1974).

[13] L. G. Stucki, "Automatic Generation of Self Metric Software," *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability*, 94-100 (May 1973).

[14] J. C. Ling, "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University (1969).

[15] R. S. Fabry, "Dynamic Verification of Operating System Decisions," *Comm. ACM*, 16, No. 11, 659-68 (November 1973).

[16] N. Chomsky, "On Certain Formal Properties of Grammars," *Information and Control*, 2, 137-67 (1959).

[17] D. Teichroew, "A Survey of Languages For Stating Requirements For Computer Based Information Systems," *Proceedings of 1972 Fall Joint Computer Conference*, 11, 1203-224.

[18] J. W. Young, Jr., "Non-Procedural Language: A Tutorial," *7th Annual Tech Meeting South California Chapter ACM* (March 23, 1965).

[19] J. E. Sammet, *Programming languages: history and fundamentals* (New Jersey: Prentice-Hall, Inc., 1969).

[20] P. Naur et al., "Revised Report on Algorithmic Language ALGOL 60," *Comm. ACM*, 6, 1 (January 1963).

[21] "System Decomposition Technology Capability Description," Computer Sciences Corporation, CDRL A004, DASG60-75-C-0125 (November 1975).

[22] "System Decomposition Technology Description," Computer Sciences Corporation, CDRL A005, DASG60-75-C-0125 (January 1976).

A SOFTWARE PHYSICS ANALYSIS OF AKIYAMA'S DEBUGGING DATA

Y. Funami
Business Administration Division,
The Fuji Bank, Tokyo, Japan

and

M. H. Halstead
Department of Computer Science,
Purdue University, West Lafayette, IN

The field of Software Physics includes a simple hypothesis relating measurable properties of computer programs to E, the number of mental discriminations performed in their production. This relation was applied to an excellent set of System Program Debugging Data published by Akiyama 1971. The result yielded surprising agreement, (coefficient of correlation of 0.98) between E as calculated from Akiyama's Data and B, the number of bugs reported in each of the nine modules of the system. Also of interest, the estimate of programming time calculated from the equation was 8 man-months, compared with the 100 man-months reported

I. INTRODUCTION

It is probably obvious that B, the number of errors that a programmer might make implementing any given algorithm in any given programming language, depends upon the total number of opportunities for making an error. Until recently, it has also been equally obvious that there was little reason to expect that such a basic quantity even existed, and even less reason to suspect that it could be measured. Recent discoveries in an area of Natural Science called Algorithm Dynamics or Software Physics [6, 8, 9, 10, 14], however, include a simple hypothesis which relates a set of measurable parameters of a program to the total number of elementary mental discriminations required to create that program. A few experiments on Programmer Productivity [5, 7, 11, 13] have suggested that the hypothesis successfully accounts for the combined effects of program volume and program difficulty.

None of the reported studies have specifically addressed the application to program bugs. Yet if the hypothesis is in reasonable agreement with reality it yields the total number of elementary mental discriminations required in writing a program, and this must also be the total number of possibilities for making an erroneous discrimination. In the following sections we will reproduce the hypothesis and apply it to an independent set of data presented to the Lubjana Conference by Akiyama in 1971.

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976

II. THE DATA BASE

Akiyama [1] has published a careful study of the number of bugs which occurred in programming each of the nine modules of a large software system called SAMPLE. In addition to data on the size of each module, Akiyama included counts of the number of decisions D, and the number of subroutine calls J, in each module. These observations, copied directly from his table, are reproduced in Table I. In the case of Module MC, 53 bugs were reported before machine runs were obtained, and these have been included.

TABLE I, Akiyama's observations.

PROGRAM MODULE		MA	MB	MC	MD	ME	MF	MG	MH	MX
Program Steps	(S)	4032	1329	5453	1674	2051	2513	699	3792	3412
Decisions	(D)	372	215	552	131	315	217	104	233	416
Calls	(J)	283	44	362	130	197	186	32	110	230
Number of Bugs	(B)	102	28	146	26	71	37	16	50	80

In presenting his data, Akiyama noted a most interesting and important feature. This finding was that the number of bugs was not as closely correlated with the sizes of the modules as it was with the sum of decisions plus subroutine calls. In fact, he reported that the coefficient of correlation between B and S was 0.83 and that the correlation between B and D plus J was 0.92. In other words, a rather crude measure of program complexity was statistically more significant in explaining the occurrence of programming bugs than program length alone.

The amount of programming time to complete individual modules was not given, but for the complete SAMPLE system, Akiyama reported that "man-hours needed for production and inspection were about 100 man-months."

III. THEORETICAL BACKGROUND

Software Physics begins by defining four basic metrics which can be obtained from the implementations or listing of any algorithm in any language. These are:

(1) η_1 = Count of the number of different operators appearing in the program
(2) η_2 = Count of the number of different operands appearing in the program
(3) N_1 = Cumulative count of all usages of operators in the program
(4) N_2 = Cumulative count of all usages of operands in the program.

Operands are defined simply as variables or constants. Operators include arithmetic symbols, delimiters, function names, subroutine names, and GOTO statements. Each pair above may be combined to give

$$\eta = \eta_1 + \eta_2 = \text{Vocabulary} , \tag{1}$$

$$N = N_1 + N_2 = \text{Length} . \tag{2}$$

It was discovered [6], and later independently confirmed [2, 4] that the relation

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 ,$$

while clearly not a mathematical identity, nevertheless holds over a large range of program sizes and programming languages.

Additional properties of programs may be obtained in terms of the four basic metrics. These include the volume V, the potential volume V^*, and the level L. The volume is obtained by multiplying the number of items appearing in a program by the number of bits which would be required to provide a unique pointer to each item in its vocabulary, or

$$V = N \log_2 \eta . \tag{3}$$

Similarly, the potential, or minimum possible volume of an algorithm as expressed in the highest conceivable language, must have a volume based upon only two operators (the name of the function and a grouping symbol) and the number of conceptually unique operands η_2^*, required for input/output parameters. Since there will be no repetitions, Eq. (4) becomes, for potential volume

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*) . \tag{4}$$

The program level is then defined as the ratio of its potential volume to its actual volume, or

$$L = V^*/V . \tag{5}$$

Since V^* is independent of the language in which an algorithm is written, the product $L \times V$ is also language independent. Now it has also been shown [6] and confirmed [3] that a good approximation to level is given by

$$L = \frac{2}{\eta_1} \frac{\eta_2}{N_2} . \tag{7}$$

Building upon this background, the next section will outline the steps followed in obtaining the programming effort hypothesis.

IV. THE EFFORT HYPOTHESIS

Based upon the quantities and relationships of the previous section, the following steps yield a measure of the number of basic or elementary mental discriminations performed in the task of implementing any given algorithm in any given language

(1) A program consist of N nonrandom select' from η
(2) Each nonrandom selection must require a search among the η items
(3) This search can be approximated by a binary search
(4) Each binary search requires $\log_2 \eta$ comparisons
(5) Therefore, a program is generated by making $N \log_2 \eta$ comparisons
(6) The volume V, is therefore a count of the number of mental comparisons required to generate a program
(7) The number of elementary or basic mental discriminations required to complete one comparison is a measure of the difficulty of the task
(8) The level L, is the reciprocal of the difficulty
(9) Then E, the count of elementary mental discriminations required to generate a program, is given by

$$E = V/L \ . \tag{8}$$

In the next section, the simple hypothesis contained in Eq. (8) will be shown to be in agreement with Akiyama's data.

Now it has previously been shown [5, 10, 11] that E can be converted from units of elementary mental discriminations to units of time by dividing by the Stroud number S, where S is the number of basic discriminations per unit time for the human brain. According to the psychologist John Stroud [12], this value lies "between 5 and 20 or a little less" per second. For concentrating (non-time sharing) programmers, fluent in their programming languages and provided with nonprocedural problem statements, a value of 18 discriminations per second has been used successfully in all of the few studies reported to date [5, 7, 11, 13].

V. COMPARING DATA WITH HYPOTHESIS

While Akiyama's data does not include the four basic software parameters directly, it supply observations from which they may be estimated. If we assume that each of the S machine language steps includes one operator and one operand, then

$$N_2 = S \ , \tag{9}$$

$$N = 2S \ . \tag{10}$$

Now the number of unique operators η_1, is composed of three classes of operators. The first is the number of distinct operators used from the machine's repertoire of tuctions. For large programs, this component may be roughly approximated as an octal hundred, or 64. Second is the number of distinct operations provided by functions or subroutines. This component should correspond to item J in Table I. Finally, each transfer to a unique location has been shown by Bulut [3] to contribute directly to η_1. the number of transfers implied by item D in Table I do not each involve transfer to a *unique* location, only a fraction, perhaps one third, should contribute to η_1. We then have, roughly

$$\eta_1 = D/3 + J + 64 \ . \tag{11}$$

At this point, we need only an estimate of η_2 to be able to calculate E. Since we have estimates of N and η_1, the length relation, Eq. (3), will yield solutions for η_2.

Using Eq. (10) for N, Eq. (9) for N_2, Eq. (11) for η_1, Eq. (3) for η_2, Eq. (4) for V, Eq. (7) for L, and Eq. (8) for E, the data of Table I yields the results shown in Table II.

TABLE II. Software physics parameters derived from Table I.

MODULE	MA	MB	MC	MD	ME	MF	MG	MH	MX
N	8064	2658	10906	3348	4102	5026	1398	7584	6824
N_1	4032	1329	5453	1674	2051	2513	699	3792	3412
η_1	471	180	610	231	366	322	131	252	433
η_2	442	176	574	201	138	287	76	603	357
E (Millions)	170.3	15.3	322.6	28.2	100.2	65.5	6.5	58.5	135.9

Arranging the modules in order of increasing number of bugs B, and comparing with the software hypothesis for E, we have the results given in Table III.

TABLE III. Errors vs discriminations.

PROGRAM MODULE	B	E (MILLIONS)
MG	16	6.5
MB	18	15.3
MD	26	28.2
MF	37	65.5
MH	50	58.5
ME	71	100.2
MX	80	135.9
MA	102	170.3
MC	146	322.6

Coefficient of Correlation = 0.982

While it remains as a task for the future to obtain the actual number of bugs from theory, the coefficient of correlation of 0.982 indicates that a large part of the variation between modules in this sample has been explained.

On the other hand, it is already possible to use values of E to obtain estimates of programming time, by dividing by the Stroud number. As mentioned earlier, a value of S of 18 discriminations per second has been used successfully in earlier studies. If we take a man-month as consisting of four and one-sixth man weeks of 40 hours each, then it contains 600,000 seconds. At 18 discriminations per second, we have 10.8 million discriminations per man-month. Summing the nine modules gives a total of 903 million discriminations for the implementation of the complete system. E/S is therefore 903/10.8 or 84 man-months, which is in reasonable agreement with Akiyama's [illegible] "about 100 man-months."

While it much to expect that the software hypothesis will explain all future experiments with the precision found here, it would seem only prudent to suggest that further experiments should be encouraged.

ACKNOWLEDGEMENT

A part of this research was sponsored by NSF Grant GJ-31572.

REFERENCES

[1] F. Akiyama, "An Example of Software System Debugging," *Proceedings of the IFIP Congress*, 353–58 (1971).

[2] R. Bohrer, "Halstead's Criteria and Statistical Algorithms," *Proceedings of the Eighth Computer Science/Statistics Interface Symposium*, Los Angeles (February 1975).

[3] N. Bulut, "Invariant Properties of Algorithms," Ph.D. Thesis, Purdue (August 1973).

[4] J. Elshoff, "Measuring Commercial PL/1 Programs Using Halstead's Criteria," GM Research Report (October 1975). Also: *ACM SIGPLAN Notices*, 11, 5 (May 1976).

[5] R. D. Gordon and M. H. Halstead, "An Experiment Comparing Fortran Programming Times with the Software Physics Hypothesis," CSD TR 167 (October 1975).

[6] M. H. Halstead, "Natural Laws Controlling Algorithm Structure?" *ACM SIGPLAN Notices*, 7, 2 (February 1972).

[7] ——, "A Theoretical Relationship between Mental Work and Machine Language Programming," Purdue, CSD TR 67 (May 1972).

[8] —— and R. Bayer, "Algorithm Dynamics," *Proceedings of the ACM National Conference*, Atlanta (1973).

[9] ——, "Language Level, A Missing Concept in Information Theory," *ACM SIGME Performance Evaluation Review*, 2, 1 (March 1973).

[10] ——, "Software Physics: Basic Principles," IBM Research Report, R. J. 1582 (May 1975).

[11] ——, "Toward a Theoretical Basis for Estimating Programming Effort," *Proceedings of the ACM Annual Conference*, Minneapolis (1975).

[12] J. M. Stroud, "The Fine Structure of Psychological Time," *Annuals of New York Academy of Sciences*, 623–31 (1966).

[13] P. Zislis, "An Experiment in Algorithm Implementation," Purdue, CSD TR 96 (June 1973).

[14] ——, "Semantic Decomposition of Computer Programs: An Aid to Program Testing," *ACTA Information*, 4, 245–69 (1975).

EFFECT OF MANPOWER DEPLOYMENT AND BUG GENERATION ON SOFTWARE ERROR MODELS

M. L. Shooman and S. Natarajan
Department of Electrical Engineering,
Division of Computer Science, Polytechnic Institute of New York, Farmingdale, NY

Several previous models in the literature have discussed how the number of errors in a large software system is related to the rate of error removal. In 1971 Shooman, Jelinski, and Moranda proposed similar probabilistic models for the removal rate of software errors during software development. The models proposed by Shooman were based on error data on seven different large operating systems and application programs collected by Hesse and the models also fit the data of Akiyama which was collected on small programs. Expressions for the number of remaining errors as the software undergoes debugging were formulated and additional assumptions were made to relate the number of residual errors to the operational system reliability.

One of the key assumptions in the above models was that the sum of the errors removed and the remaining in the program is a constant. Thus, if we can *estimate* the initial number of errors in the system at the start of debugging and keep careful records of those removed, we have a good estimate of the number of remaining errors. In 1973 Shooman described a test procedure for estimating the initial number of errors.

In this paper we add a major refinement to the above models by introducing the possibility of error generation during debugging. A generated error is due to one of two causes: (1) a bug whose correction is invalid and further debugging on the same statements is essential, and (2) a new bug which is generated as the result of the correction of a different error. The error generation terms are modeled in several different ways, proportional to the number of detected errors, corrected errors, number of remaining errors, or some function of these effects. The correction rate is assumed to be a function of the manpower deployed on the project, thus, one can use the model to investigate optimum manpower deployment strategies.

I. INTRODUCTION

The software field is nearly two decades old and the problems and costs associated with the debugging phase of software are especially acute. It is important to understand the aspects of bug generation and removal inherent in most computer programs. In this paper we discuss the dynamics of error generation and removal as debugging proceeds.

Earlier work [1-3] in this field was done with the assumption that the total number of errors (i.e., sum of those corrected and those remaining) in a program remains constant throughout the debugging process. This leads to the assumption that at some stage of debugging we will have removed all the errors. In practice the contrary is true. Thus, a

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976.

certain amount ... generation of errors is associated with any debugging procedure. Therefore, the total number of errors does not remain a constant. Some of the ways in which errors may be generated are:

(1) The correction of a bug may work locally only (i.e., the global aspects of the error still remain)
(2) A typographical error may arise invalidating the result of bug correction
(3) The correction is based upon faulty analysis, thus complete bug removal is not accomplished
(4) The correction is accomplished; however, it is accompanied by the creation of a new error
(5) Errors which are detected but not corrected act in many ways like generated errors.

During development we are faced with two classes of changes in a program—those due to changes in the specifications (design) and those necessary to correct software errors. Although both classes of changes are important, this paper only addresses the changes needed for error correction. In evaluating the constants of the models from data, it is important to be able to differentiate between these two effects.

II. EXPERIMENTAL DATA ON DEBUGGING

Before we can hypothesize an error model, it is advisable to investigate some of the experimental results that are available to us. Shooman and Bolsky [6] report the results of an experiment on error data collection conducted at Bell Laboratories. A program of 4000 machine language words was studied during the debugging phase. There were 45 total errors out of the 4000 machine language words which represents approximately 1% of the total lines of code. This data is in agreement with previous error data on large programs in the literature [7]. One of the outcomes of this experiment is a record of the time taken to remove each error. It was found that except for a few errors, the time expended in removing errors was approximately the same. An obvious conclusion from this result is that the rate of error correction is a constant.* Initial work on the extension of a basic error model [1–4] was attempted based on the assumption of constant error correction rate and was found to lead to anomalous results. These models as well as new models which eliminate the anomalies are discussed in this paper.

Akiyama [8] reports another study on error data on small programs, at Fujitsu Limited in Tokyo. The program called SAMPLE consists of seven modules, and was programmed in FASP, the assembler language. In all these modules Akiyama observed a decreasing trend in error correction rate towards the later part of debugging. The cumulative error curves were observed to rise rapidly in the beginning and decrease in slope to nearly horizontal asymptotes towards the end. Calculation of the cumulative error curves from the data in [1] displays a similar behavior. We will deal with error models which correspond to the above experimental results.

*For other interpretations of this result see [6].

III. MODEL FOR GENERATED ERRORS

Depending upon the efficiency of debugging the error generation rate could be greater than, equal to, or less than the error correction rate. The resulting error behavior in the three cases is shown qualitatively in Fig. 1 (Fig. 1 (a) is a particular case where there is no generation) [3]. The time τ_1 is when debugging stops. Note that the number of errors remaining is greater than 0 in each case.

We wish to develop a set of equations which will describe the above cases. We begin by writing a difference equation for the number of errors in the program [10]

$$\begin{aligned}\text{Errors present at time } \tau_i = {} & [\text{errors present at time } \tau_{i-1}] + \\ & + [\text{errors generated in the interval } (\tau_i - \tau_{i-1})] - \\ & - [\text{errors removed in the interval } (\tau_i - \tau_{i-1})]\end{aligned}$$

If we let:

(1) $n_g(\tau_i, \tau_{i-1})$ = number of errors generated in the interval $(\tau_i - \tau_{i-1})$
(2) $n_d(\tau_i, \tau_{i-1})$ = number of errors detected in the interval $(\tau_i - \tau_{i-1})$
(3) $n_c(\tau_i, \tau_{i-1})$ = number of errors corrected in the interval $(\tau_i - \tau_{i-1})$,

then the number of errors remaining in a program at time τ_i, $n(\tau_i)$, is given by the following difference equation

$$n(\tau_i) = n(\tau_{i-1}) + n_g(\tau_i, \tau_{i-1}) - n_c(\tau_i, \tau_{i-1}). \tag{2}$$

Conversion of the above difference equation to a differential equation is performed by grouping terms, dividing both sides by $(\tau_i - \tau_{i-1}) \equiv \Delta\tau$, and taking limits

$$\lim_{\Delta\tau \to 0} \left[\frac{n(\tau_i) - n(\tau_{i-1})}{\Delta\tau}\right] = \lim_{\Delta\tau \to 0} \left[\frac{n_g(\tau_i, \tau_{i-1})}{\Delta\tau}\right] - \lim_{\Delta\tau \to 0} \left[\frac{n_c(\tau_i, \tau_{i-1})}{\Delta\tau}\right]$$

The left-hand side of Eq. (2) is recognized as the rate of change of errors remaining, i.e., the derivative of n with respect to τ. The right-hand side is composed of two terms which as the limit is approached, become the rates of error generation and correction respectively. The notation for error rates is given as follows:

(1) $r_g(\tau_i) \triangleq$ generation rate of new errors at time τ_i

(2) $r_c(\tau_i) \triangleq$ correction of at time τ_i

(3) $r_d(\tau_i) \triangleq$ detection rate of errors at time τ_i.

Using the above definitions Eq. (2) becomes

$$\frac{dn(\tau)}{d\tau} = r_g(\tau) - r_c(\tau) . \tag{3}$$

In Eq. (3) the term accounting for error generation includes all the five cases discussed in oduction which are created by debugging changes.

In a process where debugging is efficient, the generation rate is smaller than the correction rate, and the number of bugs in the system decreases. Steady state is reached when the correction rate decreases so that $\frac{dn(\tau)}{d\tau} \approx 0$ (see Figure 1 (a) and (b)). Let us hypothetically say that the correction rate is proportional to the detection rate

$$r_c(\tau) = \beta r_d(\tau) . \tag{4}$$

In the ideal case, all detected errors are immediately corrected and $\beta = 1$; however, in practice $0 < \beta < 1$, because some detected errors may be neglected and others may be incorrectly fixed. This latter effect adds to the generation rate as previously discussed.

The generation effect is more complicated to model. If we assume that the genera- rate is proportional to correction rate, then combining Eqs. (3), (4), and (5) yields

$$r_g(\tau) = \alpha r_c(\tau) , \tag{5}$$

$$\frac{dn(\tau)}{d\tau} = \alpha r_c(\tau) - \beta r_d(\tau) = \beta(\alpha - 1) r_d(\tau) . \tag{6}$$

Inspection of Eq. (6) shows that the effect under these assumptions for the normal $\ldots < \alpha < 1$ is that the number of errors decreases. However, if $\alpha > 1$, generation exceeds correction and the result illustrated in Fig. 1 (c) is obtained. If $\alpha = 0$, there is no generation, but in general $0 < \alpha < 1$ and the effect of error generation is to' oduce the effective correction rate (see case 1, Table I). From the solution we can see that for large τ, $n(\tau)$ becomes negative which leads to a physical contradiction.

We may explore a different model by assuming the detection rate is in turn propor- ti to the number of remaining errors. This leads to case 3 of Table I and $n(\tau)$ is a decreasing exponential. This result agrees with only a part of the experimental data reported in the literature [1, 8], since shapes other than decreasing exponentials have been observed for $n(\tau)$.

A different hypothesis is to assume that generation of new errors is a function of not the detection rate, but also the number of remaining errors. Clearly, the larger the number of detected errors, the more are the changes required and the probability of creating an error increases. Also, each time we make a change there is a chance that this will act with existing errors and create new errors via the interaction. Assuming gen- era is a simple product of these two functions and correction is as given in Eq. (4) (with β replaced by $\ldots$ e obtain

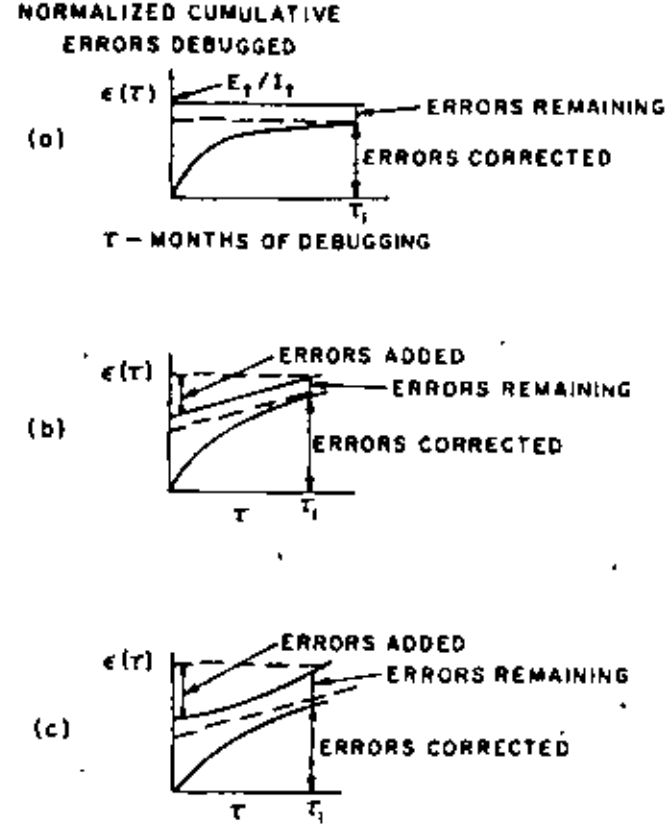


Fig. 1. Cumulative errors debugged vs months of debugging. (a) Approaching equilibrium, horizontal asymptote, no generation of new errors. (b) Approaching equilibrium, generation rate of new errors equals error removal rate. (c) Diverging process, generation rate of new errors exceeds error removal rate.

$$r_g(\tau) = a\, n(\tau)\, r_d(\tau) , \tag{7}$$

$$r_c(\tau) = b\, r_d(\tau) , \tag{8}$$

where a and b are proportionality constants.

Substituting Eqs. (7) and (8) into Eq. (3) we obtain

$$\frac{dn(\tau)}{d\tau} = \dot{n}(\tau) = a\, n(\tau)\, r_d(\tau) - b\, r_d(\tau) . \tag{9}$$

If we make the further assumption that the detection rate is a constant r_0, Eq. (9) becomes

$$\dot{n}(\tau) = a\, n(\tau)\, r_0 - b\, r_0 . \tag{ }$$

TABLE 1. Basic equation is equation (3)

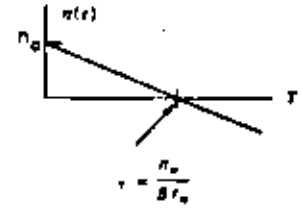
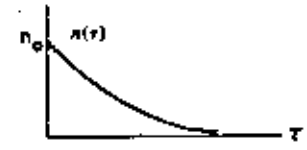
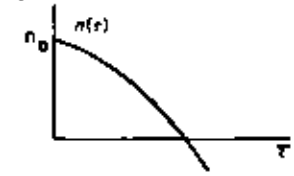
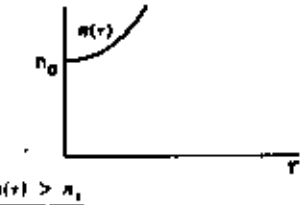
CASE	EQUATION	SOLUTION
1. Generation and correction proportional to detection $r_g(\tau) = \alpha \beta r_d(\tau)$ $r_c(\tau) = \beta r_d(\tau)$	$\dot{n}(\tau) = \beta(\alpha - 1) r_d(\tau)$	If $\alpha < 1$, $n(\tau)$ is a decreasing function If $\alpha > 1$, $n(\tau)$ is an increasing function
2. No generation $r_g(\tau) = 0$ $r_c(\tau) = \beta r_d(\tau)$ $r_d(\tau) = \text{constant} = r_0$ [correction ∼ detection; detection = constant]	$\dot{n}(\tau) = -\beta r_0$	$n(\tau) = n_0 - (\beta r_0)\tau$  $\tau = \frac{n_0}{\beta r_0}$
3. No generation $r_g(\tau) = 0$ $r_c(\tau) = \beta r_d(\tau)$ $r_d(\tau) = K_1 n(\tau)$ [correction ∼ detection; detection ∼ errors present]	$\dot{n}(\tau) = -\beta K_1 n(\tau)$	$n(\tau) = n_0 e^{-\beta K_1 \tau}$ 

TABLE 1. Basic equation is equation (3) continued.

CASE	EQUATION	SOLUTION
4. Generation proportional to product of errors present and detection $r_g(\tau) = a r_d(\tau) n(\tau)$ $r_c(\tau) = b r_d(\tau)$ [correction ∼ detection]	$\dot{n}(\tau) = [a n(\tau) - b] r_d(\tau)$ For $r_d(\tau) = \text{const.} = r_0$ $\dot{n}(\tau) = [a n(\tau) - b] r_0$	$n(\tau) = (n_0 - b/a)e^{a r_0 \tau} + b/a$ If $n_0 < b/a$ then $n(\tau)$ is an inverted exponential  If $n_0 > b/a$ then $n(\tau)$ is a growing exponential. 
5. Generation proportional to number of errors present and correction either manpower or detection limited $r_g(\tau) = a_1 n(\tau)$ $r_c(\tau) = \begin{cases} b_1 & \text{for } n(\tau) > n_1 \\ b_2 n(\tau) & \text{for } n(\tau) \le n_1 \end{cases}$	$\dot{n}(\tau) = -b_1 + a_1 n(\tau)$ for $n(\tau) > n_1$ $\dot{n}(\tau) = -b_2 n(\tau) + a_1 n(\tau)$ for $n(\tau) \le n_1$	$n(\tau) > n_1$ $n(\tau) = (n_0 - b_1/a_1)e^{a_1 \tau} + b_1/a_1$ (same form as case 4) $n(\tau) \le n_1$ $n(\tau) = n_1 \exp[(a_1 - b_2)(\tau - \tau_1)]$ (i) If $1 < b_2/a_1$, then $n(\tau)$ decays exponentially as shown in Figure 3 (b). (ii) If $1 > b_2/a_1$, then $n(\tau)$ oscillates as shown in Figure 3 (c).

The above differe equation be readily solved by taking Laplace transforms or by differential equation yielding

$$n(\tau) = (n_0 - b/a)\, e^{a\, r_0 T} + b/a \ , \qquad (11)$$

where $n(\tau = 0) \equiv n_0$.

The behavior of Eq. (11) depends upon the relative values of n_0 and b/a. Since the probability of n_0 being exactly equal to b/a is very low, we are left essentially with two possibilities. If $n_0 > b/a$, then $n(\tau)$ builds up exponentially. If $n_0 < b/a$ the number of errors decreases and becomes negative for large τ. The two cases are shown in Fig. 2, and case 4, Table I.

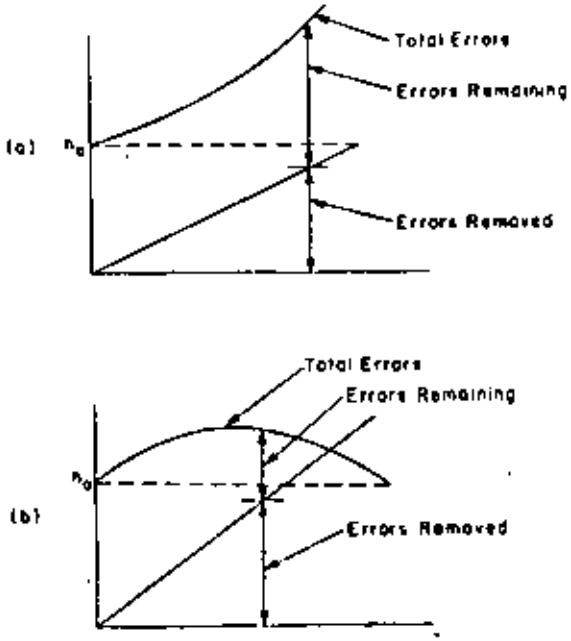


Fig. 2. The model developed under the assumptions of case 4, Table I [4]. (a) The case where $n_0 > b/a$. (b) The case where $n_0 < b/a$.

The fact that the number of errors can become negative (as shown in Fig. 2 (b)) has no physical meaning. Thus, this model also leads to a contradiction. Either the initial assumption and the model are only valid for a short time period, or our initial assumptions were wrong. In fact we have assumed several models for error generation and have rejected the results Eqs. (6) and (11) because the remaining errors went negative or decreased in an exponential fashion. We now turn toward models for the error correction rate which may be more realistic, so as to obtain an expression for $n(\tau)$ which does not violate physical reasoning and fits the experimental data.

A different form for the correction term in Eq. (3) can be developed if we assume the rection is proportional to the detection which in turn is dependent on the debugging manpower. A particular form for the manpower dependency is discussed in the next section.

IV. MANPOWER LIMITED MODEL

As reported in [1, 8] a decreasing trend in error correction rate has been observed towards the later part of debugging. On the other hand, Shooman and Bolsky [6] observed a constant correction rate. Also, discussions with program managers have the conclusion that correction rate is sometimes manpower limited. Thus, we postulate new model where the correction rate remains constant during the early stage of debugging (manpower limited). We assume that later in the program another stage is reached where the correction rate is proportional to the number of remaining errors. During both these correction stages we assume the error generation rate is proport to the product of the number of remaining errors and the number of detected errors as Equation (9). The transition from the early stage model of error correction to the late stage model may be considered to occur at a critical value of the remaining number of bugs which we call n_1.

These assumptions lead to

$$r_g = k_1\, n(\tau)\, r_d(\tau) \qquad \text{for all } n(\tau) \ , \qquad (1$$

$$r_c = k_2\, r_d(\tau) \qquad \text{for } n(\tau) > n_1 \quad \text{[Region 1]} \qquad (1$$

$$r_c = k_3\, n(\tau)\, r_d(\tau) \qquad \text{for } n(\tau) \le n_1 \quad \text{[Region 2]} \qquad (1$$

If we make the further assumption that $r_d(\tau)$ is a constant we obtain

$$r_g = a_1\, n(\tau) \ , \qquad (1$$

$$r_c = b_1 \qquad \text{for } n(\tau) > n_1 \ , \qquad (1$$

$$r_c = b_2\, n(\tau) \qquad \text{for } n(\tau) \le n_1 \ , \qquad (1$$

where

$$a_1 = k_1\, r_d(\tau) \ ,$$
$$b_1 = k_2\, r_d(\tau) \ ,$$
$$b_2 = k_3\, r_d(\tau) \ .$$

Substituting Eqs. (14) and (15a) into Eq. (3) we get for the early phase where $n(\tau) >$ (called Region 1)

$$\frac{dn(\tau)}{d\tau} = a_1 n(\tau) - b_1 \quad . \tag{16}$$

A solution of Eq. (16) is of the same form as Eq. (11)

$$n(\tau) = (n_0 - b_1/a_1) e^{a_1 \tau} + b_1/a_1 \quad . \tag{17}$$

If $n_0 > b_1/a_1$ the debugging is out of control and Eq. (17) indicates that the errors buildup exponentially with time.* On the other hand if $n_0 < b_1/a_1$ the correction process is efficient and the errors reduce with time. Once the errors fall to the critical value n_1, a transition takes place in the error correction rate and substitution of Eqs. (14) and (15b) into Eq. (3) yields

$$\frac{dn(\tau)}{d\tau} = a_1 n(\tau) - b_2 n(\tau) \quad . \tag{18}$$

Letting the time elapsed in reducing the number of errors to n_1 be τ_1, a solution of Eq. (18) yields

$$n(\tau) = n_1 \exp [(a_1 - b_2)(\tau - \tau_1)] \quad . \tag{19}$$

The conditions under which the transition occurs is explained as follows: As the errors decrease, the number of men employed is also reduced [9]. Since we intuitively feel that the later bugs are harder to fix, even if we maintain the ratio of the number of errors to number of men constant, we will observe a decreasing correction rate. (Note: This contradicts some of the results of [6].)

Equation (19) in itself gives rise to two cases depending upon whether $1 > b_2/a_1$ or $1 < b_2/a_1$. If $1 < b_2/a_1$, then $n(\tau)$ decays exponentially to zero as τ goes to infinity. On the other hand, if, $1 > b_2/a_1$ then $n(\tau)$ increases exponentially, thereby re-entering Region 1. A physical explanation for why the switch in regions may occur is, if the manager removes some of the personnel on the project prematurely, he thereby decreases b_2. If he later notices that the errors are increasing, some personnel are brought back and a transition to Region 2 will soon occur. We thus can have an oscillating model. It is convenient to categorize the models as:

(Case 1) The unstable model
(Case 2) The controlled model
(Case 3) The oscillatory model.

The three cases are shown in Fig. 3 and summarized in case 5, Table I. We will analyze each case separately.

*Discussions with experienced software managers have verified that on rare occasions debugging does go out of control and either the program is scrapped and rewritten or a new team of "super debuggers" is brought in.

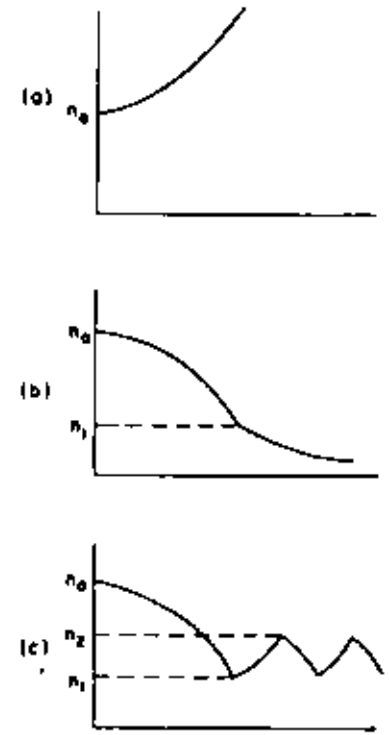


Fig. 3. Remaining errors plotted as a function of months of debugging. (a) Unstable model, errors build up indiscriminately. (b) Controlled model, debugging is efficient. (c) Oscillatory model.

A. Case 1

In Fig. 3 (a) the errors are seen to diverge exponentially. If the software personnel are not sufficiently experienced, a more experienced team could be brought in to replace the current one. At this point let us say, the errors have built up to n' at time τ'. The new team establishes different values for b_1 and a_1 (viz. b_1' and a_1' such that $n' < b_1'/a_1'$).

The equation describing error correction is

$$n_c(\tau) = b_1 \tau \qquad \text{for } \tau < \tau' \quad ,$$

while $n(\tau)$ is described by Equation (17). The total number of errors at any time is the sum of those remaining and those corrected, resulting in

$$n_t(\tau) = n(\tau) + n_c(\tau) \quad .$$

For $\tau > \tau'$ the error equation can be written using Eq. (17) resulting in

$$n(\tau) = (n' - b_1'/a_1') \exp \left[a_1' (\tau - \tau') \right] + b_1'/a_1' \quad .$$

The error correction is given by

$$n_c(\tau) = b_1 \tau' + b_1'(\tau - \tau') \tag{23}$$

Equations (22) and (23) are applicable between τ' and τ_1 where τ_1 marks the point at which the errors fall to the critical value n_1. Beyond τ_1 the error equation is defined by Equation (19). To compute the number of errors corrected we can rewrite Eq. (15b) as

$$\frac{dn_c(\tau)}{d\tau} = b_2\, n(\tau) ,$$

which results in

$$n_c(\tau) = \int b_2\, n(\tau)\, d\tau ,$$

or

$$n_c(\tau) = b_2 n_1 \int \exp\left[(a_1' - b_2)(\tau - \tau_1)\right] d\tau ,$$

or

$$n_c(\tau) = \frac{b_2 n_1}{a_1' - b_2} \exp\left[(a_1' - b_2)(\tau - \tau_1)\right] + C . \tag{24}$$

Using the boundary condition that $n_c(\tau_1) = b_1 \tau' + b_1'(\tau_1 - \tau')$ which is true from Eq. (23) the number of corrected errors is written down as

$$n_c(\tau) = b_1\tau' + b_1'(\tau_1 - \tau') + \frac{b_2 n_1}{a_1' - b_2} \left\{ \exp\left[(a_1' - b_2)(\tau - \tau_1)\right] - 1 \right\} . \tag{25}$$

B. Case 2

The initial stage here is described by Eq. (17) with $n_0 < b_1/a_1$. The correction is governed by Equation (20). The errors fall exponentially and a transition occurs at τ_1 beyond which the error model is described by Eq. (19) with $1 < b_2/a_1$. The correction curve beyond τ_1 is described by Eq. (24) with the only difference of a_1 replacing a_1' and the boundary condition being

$$n_c(\tau_1) = b_1 \tau_1 .$$

The correction curve is therefore described by

$$n_c(\tau) = b_1\tau_1 + \frac{b_2 n_1}{a_1 - b_2} \left\{ \exp\left[(a_1 - b_2)(\tau - \tau_1)\right] - 1 \right\} \text{ for } \tau \geq \tau_1 . \tag{26}$$

C. Case 3

In the case of the oscillatory model, a substantial number of errors have been removed, and if the oscillations can be broken, the system debugging can be promptly completed. One of the reasons for b_2 being less than a_1 could be that the debugging personnel were removed too early. If the transition had occurred at a lower value of n_1 the pattern might have been the same as of case 2. Therefore, once the process goes back to Region 1, the software manager may increase the debugging personnel. If this is effective we may stay in Region 2 till completion of debugging; however, if personnel are removed again we may restart the oscillatory behavior.

Here the initial behavior of errors is given by Eq. (17) with $n_0 < b_1/a_1$, while the correction is described by Equation (20). Upon a transition at $n = n_1$ the model obeys Eq. (19) with $1 > b_2/a_1$. After transition the correction is described by Equation (26) $n(\tau)$ now increases to n_2 at time τ_2 at which point another transition (this happens due to personnel being brought back) occurs bending the error curve downward. Under these conditions the model is described by an equation similar to Equation (17). The equation is

$$n(\tau) = (n_2 - b_1/a_1) \exp\left[a_1(\tau - \tau_2)\right] + b_1/a_1 , \tag{27}$$

using Eq. (26) the correction curve can now be written as

$$n_c(\tau) = b_1\tau_1 + \frac{b_2 n_2}{a_1 - b_2} \left\{ \exp\left[(a_1 - b_2)(\tau_2 - \tau_1)\right] - 1 \right\} + b_1(\tau - \tau_2) \text{ for } \tau \geq \tau_2 . \tag{28}$$

The errors now fall until a critical point defined by $n = n_3$* at τ_3 when another transition takes place. Below n_3 the error equation is described by an equation similar to that of Equation (19). The equation is

$$n(\tau) = n_3 \exp\left[(a_1 - b_3)(\tau - \tau_3)\right] \quad \text{for } \tau \geq \tau_3 , \tag{29}$$

once again Eq. (26) can be used to write an equation which describes the correction beyond τ_3

$$n_c(\tau) = \frac{b_3 n_3}{a_1 - b_3} \left\{ \exp\left[(a_1 - b_3)(\tau - \tau_3)\right] - 1 \right\} + n_c(\tau_3) , \tag{30}$$

where

$$n_c(\tau_3) = b_1\tau_1 + b_1(\tau_3 - \tau_2) + \frac{b_2 n_2}{a_1 - b_2} \left\{ \exp\left[(a_1 - b_2)(\tau_2 - \tau_1)\right] - 1 \right\}$$

*n_3 is set smaller than n_1 to avoid oscillations. Under these conditions $R_{eff} = \frac{1}{\ldots}$ some where $\ldots < n_3$. The software team now establishes a new value for b_1, $\ldots$ $1 < b_1/a_1$

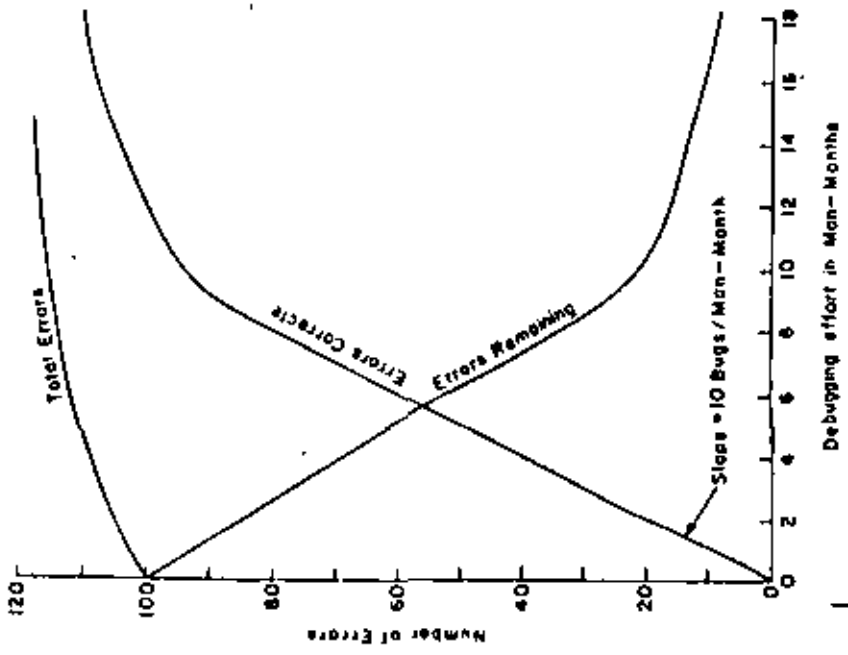


Fig. 5. Case 2 - the controlled model

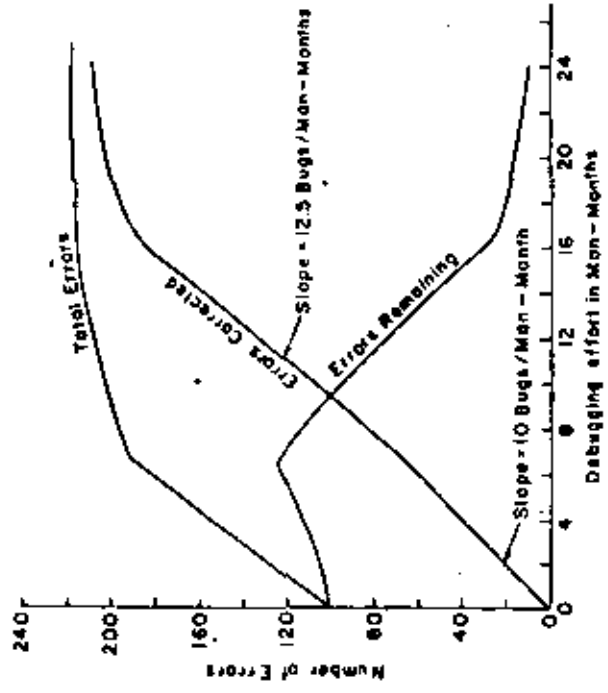


Fig. 4. Case 1 - unstable model controlled subsequently.

which is obtained by substituting τ_3 for t in Equation (28). The cases are shot Figures 4, 5, and 6. An initial error quantity of $n_0 = 100$ has been assumed in all ca

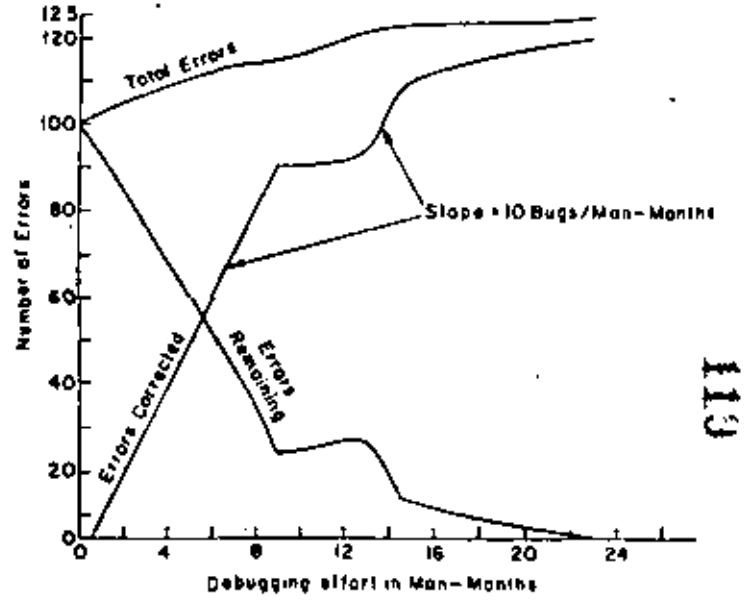


Fig. 6. Case 3—the oscillatory model. The oscillations are controlled after the first bump

A PL/1 computer program has been written which plots the number of remaining errors, the cumulative corrected errors, and the total number of errors. The calculate points for the curves in Figs. 4, 5, and 6 were computed using this program.

V. SUMMARY

The hypothetical models developed in this paper appear in summary form in Tabl Many of the features of these models agree with the data reported in the literature an the experiences of program managers; however, no detailed experimental data is avail A critical analysis of these models must await further experimental data collection co cerning the dynamics of the error removal-generation process.

REFERENCES

[1] M. L. Shooman, *Probabilistic Models for Software Reliability Prediction* Co ... *Statistical Methods for the Evaluation of Computer System Performance*, ... (New York: Academic Press, 1971)

[2] Z. [illegible] and P. B. Moranda, "Software Reliability Research," *Statistical Computer Performance Evaluation* (New York: Academic Press, 1972).

[3] M. L. Shooman "Operational Testing and Software Reliability Estimation During Program Development," *1973 IEEE Symposium on Computer Software Reliability*, New York City (April 30–May 2, 1973).

[4] ——, et al, unpublished memoranda on Error Generation Models, Bell Laboratories (March 1973).

[5] J. D. Musa, "A Theory of Software Reliability and Its Application," *IEEE Trans. Software Eng.*, SE-1, No. 3 (September 1975).

[6] M. L. Shooman and M. I. Bolsky, "Types, Distributions, and Test and Correction Times for Programming Errors," *Proceedings of the 1975 International Conference on Reliable Software*, Los Angeles (April 21–23, 1975).

[7] J. Dickson, J. Hesse, A. Kientz, and M. Shooman, "Quantitative Analysis of Software Reliability," *1972 IEEE Annual Reliability Symposium Proceedings* (January 1972).

[8] F. Akiyama, "An Example of Software System Debugging," *IFIP Congress 1971*, Ljubljana, Yugoslavia (August 1971).

[9] F. P. Brooks, Jr., "How does the Project Get to Be a Year Late?—One Day at a Time," *Datamation* (December 1974).

[10] S. Natarajan and M. Shooman, unpublished report on Error Generation - Removal Models, Polytechnic Institute of New York (Fall 1975).

AN EXPERIMENT IN AUTOMATIC QUALITY EVALUATION OF SOFTWARE

S. J. Amster, E. J. Davis, and B. N. Dickman
Bell Laboratories, Holmdel, Whippany, and Murray Hill, NJ (respectively)
and
J. P. Kuoni
American Telephone and Telegraph, New York, NY

The purpose of the experiment was to see if it was possible to provide a useful tool to aid in the improvement and automatic measurement of the quality of computer programs. Such a tool was wanted because of the large number (over 1000) of programs involved, and because of a desire to obtain quantitative measures of quality. There were five subjective quality definitions considered. The first two dealt with the extent to which reductions in object code could be made via simple transformations or a complete restructuring of the program. The next two consisted of the extent to which reductions in the number of source statements could be made via simple transformations or a complete restructuring of the program. The last was the ranked clarity of the program source. The principal method used was the manual grading of a sample, extraction of quantifiable independent variables, and the use of regression analysis to derive prediction formulas. Results included some tentative quality prediction formulas, correlations among the independent and dependent variables (e.g., GOTO and clarity), observations about programming, and a host of newly-generated questions. There are indications that for two large program populations, we have derived a useful tool for automatically differentiating between good and bad programs.

1. INTRODUCTION

A. Purpose

The purpose of the experiment was to see if it was possible to provide a useful tool to aid the improvement and automatic measurement of the quality of computer programs. The improvement of programs can be aided if automatic code scanning heuristic can be produced which can distinguish programs which are candidates for improvement specified areas. A tool for the measurement of the quality of programs can be provided if the heuristics are able to rank programs. Of course, the latter is more difficult. Seven criteria for quality have been specified, and heuristics for accomplishing both objectives have been successfully derived.

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976.

There were two reasons why such a study was desirable. The first was the large um of programs (over 1000) involved, making human grading and selection of pro- rams ssible. Secondly, there was the desire to obtain quantitative definitions of uality. What is there about a program that we can measure that makes us say that it is ood or bad?

C. Experiment Overview

The experiment was performed in two steps: development; and application of the rediction formulas. In the first step, a random sample of ETC [4] control programs was raded independently by several programmers according to various criteria for quality. or each criterion the gradings (dependent variables) were discussed, and a consensus ading was reached. Concurrently, tapes of the listings for this sample and the resulting bject code expansions were input into a data extraction program. Certain quantifiable ndependent variables (referred to as X variables) were constructed from the extracted ata. Statistical procedures, primarily multiple regression analyses, were employed to erive formulas to be used to predict the quality of a program.

A second random sample was selected, and a consensus grading was determined. This rading and the values predicted by the previously derived formulas were compared and ere found close enough to validate, at least tentatively, the prediction formulas for the te population.

The second stage of the experiment was to analyze further the data and to apply the formulas to two large sets of programs. Numerous analyses and applications were attempted including:

(1) Intercorrelations between independent and dependent variables
(2) Quality measurements of the programs in two large systems
(3) Quality measurements of functional groups comprising the systems
(4) Comparisons of the use of structured programming with clarity.

D. Project Overview

It was felt desirable to experiment with a large homogeneous population of programs. Two large scale software systems developed by Bell Laboratories and several subcontractors provided two such populations. These two systems (called PW and MW), in total over 1000 programs and over 250,000 statements, were subject to strict change control and documentation procedures. The project for which the systems were developed is described in [12].

E. Characteristics of the Programming Language

The project programming language ETC, can be described as an extensible language in which several levels of language features exist. At the lowest level, ETC is the assembly language. At the next level, ETC provides a uniformity for the machine language, completing incomplete data paths, providing uniform register usage. At this level, ETC is still more-or-less one-to-one with the machine code, but provides more concise syntax for the machine operations by means of, for example, polymorphic operators. At the next level, machine dependence may still exist in the form of hardware register references, but ETC functions as a true compiler. At the highest level of use, ETC programs can be as machine independent as those written in PL/1.

At the highest level, ETC approximates PL/1 in control structure and FORTRAN in data structure. In addition to the control structure of PL/1, ETC has CASE, BREAK, and ITERATE statements. CASE allows a multipath branch; BREAK allows a program to exit a DO loop or group gracefully (without use of a GOTO); ITERATE causes the next iteration of a DO loop or group to begin. The data structures explicitly supported are similar to FORTRAN's, except that based variables, simple structures, and partial word variables are allowed. The base language has been described in [4].

F. Characteristics of the Programs and the Programmers

The programmer population, consisting of several hundred programmers, exhibited a wide range of abilities and experience. It is not clear that the apparent novices among the programmer population consistently wrote lower quality programs than their more experienced colleagues. In fact, for those whose prior programming experience was limited to assembly language, the interference effect of learning a second language, and the difficulty of abandoning the habit of thinking in terms of assembly language, probably contributed to the observed (unnecessary) lack of language leverage.

In an effort to insure conformance to uniform programming standards, structured programming was required for all of the PW and part of the MW populations of programs. Forbidden were the use of GOTO, IF, BREAK, and ITERATE. It is unclear why IF, BREAK, and ITERATE were forbidden. In examining a sample of programs, structured and with GOTOs, several points have been noted. First, recall that it has been shown in several papers that a program with GOTOs may be mechanically transformed into a GOTO-free program (assuming appropriate language constructs) by means of "auxiliary variables" (flags, switches) and by means of replicating blocks of code. Of course this is presented as a theoretical result; no one would call the programs thus transformed good structured programs. Unfortunately this is exactly how some programmers program to avoid the discipline of structured programming. Structured programming is encouraged to provide top-to-bottom flow of control and locality of control structure and data use. These goals are effectively defeated if programmers simulate GOTOs by setting and testing flags and switches everywhere. Maintainability is hurt by the replication of blocks of code. In fact, given the usual structured programming features, the *opportunity* for truly horrendous programming is increased. Consider the following sequence of statements

```
DO LINK1 = 0 TO 3
  DO CASE LINK1
    CASE 0
      A = 73
    CASE 1
      B = 74
    CASE 2
      C = 23
    CASE 3
      D = 55
  DOEND
DOEND,
```

which is equivalent to

```
A = 73
B = 74
C = 23
D = 55.
```

The code at each CASE was simplified here for the purpose of the example, but, otherwise is as taken from a working production program. (In the original program too there was only straight line code at each CASE.) The same construct occurred at least twice in a random sample of 14 programs. This is not to say that structured programming should not be encouraged, but only that it provides as much "leverage" for bad programming as for good. The better programmers produced good quality code by observing the spirit rather than the letter of the rules.

II. DEVELOPMENT OF THE PREDICTION FORMULAS

A. Quality Criteria

For the purposes of this study the quality criteria are disassociated from the objectives of the program designer and coder. For instance, the programmer may have as his primary objective optimization of execution time, rather than program size. Also, a program may have become degraded due to the implementation of new requirements or "one-line" changes to correct bugs. Neither the program quality objectives nor the history of the program has any bearing on the rating of a program according to a particular measure of quality. If someone needs to reduce the size of an object module, it does not matter how the "bad" programs got so large or why the programs are so large. It is left to the discretion of the user of the quality assurance tools whether programs *should* be rated according to a particular quality criterion.

The method here was to proceed from relatively objective, easily defined dependent variables to more subjective, less easily defined dependent variables, with the idea that if we did not succeed with the easily defined dependent variables, we probably would not succeed with the difficult ones. This led us to consider "static" quality criteria (e.g., program size) before dynamic criteria (e.g., execution time, which depends on the input data). Furthermore, there was an attempt to pick easily defined variables which not only were inherently useful to consider, but might correlate highly with those more difficult to define.

Five criteria of quality were considered in the study. The first dealt with the extent to which reductions in object code could be made via simple transformations of the program; the second dealt with reductions in object code that could be made by a complete restructuring. The third and the fourth criteria consisted of the extent to which reductions in the number of source statements could be made via simple transformations or a complete restructuring of the program. Finally, we considered the clarity of the programs. The first four criteria were measured in percent reduction of source or object statements; the last was measured as a rank, nominally 1-14 (best to worst). These variables are referred to later in this paper as the *dependent* variables and are identified as $Y_1, \cdots, Y_5$.

1. *Conciseness of Object Program*

The first criterion to be considered in the study was the ease of reduction of the number of machine operations, that is, where to spend your time to get the greatest reduction in number of machine operations for time spent. This quality criterion was chosen first because it is relatively easy to measure. Program size is a static quality (as opposed to execution time); it does not vary with each execution. By "reduce" is meant "perform a function-preserving transformation of the program that reduces the number of generated machine operations." Since for some bad programs little reduction could be accomplished easily, but a great reduction was possible with a complete rewrite of the program, two types of reduction were defined. An "easy reduction" is a more-or-less mechanical transformation to reduce the number of machine operations, e.g., rephrasing of DO loops, changing type declarations, simple restructuring (e.g., introducing CASE statements), avoiding unnecessary fetches and stores. A "difficult reduction" is a complete restructuring of the program to reduce the number of machine operations by redrawing the flow charts and understanding the purpose of the program. Since ETC allows one to control the generation of object code very closely if desired, it is valid to consider this as a quality criterion subject to improvement.

For example, suppose program A with 1000 machine operations and program B with 200 machine operations can each be decreased 20% by easy reductions. If program C has 1000 machine operations which can be decreased 10% by easy reductions, then A and B are of equal quality, lower than that of C.

2. *Conciseness of Source Program*

The next criteria of quality to be considered were concerned with ease of reduction of source program size, that is, where to spend your time to get the greatest reduction for time spent in the number of source statements. These criteria are defined exactly as for conciseness of object program, except with "source statements" replacing "machine operations." Conciseness of source is important to consider, because it is a prime attribute of clarity, the next criterion of quality to be considered. The example of Section I

illustrates the type of transformation made. Not only were the resulting programs more , but also clearer and smaller in object code.

An example of a transformation made to reduce source (and sometimes, depending on the type of variables involved, to reduce object) involved changing

```
A = 0
B = 0 .
```

to the iple assignment statement

```
(A,B) = 0 .
```

Another construct that occurred often was

```
IF A = 1 THEN DO
  IF B > 1 THEN DO
    .
    .
    .
  DOEND
DOEND
```

This is e concisely and clearly stated as

```
IF A = 1 AND B > 1 THEN DO
  .
  .
  .
DOEND
```

The construct

```
WHEN A < = 1 THEN;
ELSE · · · ,
```

is r stated as

```
IF A > 1 THEN · · · .
```

Finally, in the few cases where structured programming was not used, the following occurred with some frequency

```
  IF A < = B THEN GOTO C
  GOTO D
C · · ·
```


instead of (at worst)

```
IF A > B THEN GOTO D .
```

3. Clarity of Source Program

Clarity was chosen as the next criterion of quality because it would be hoped that clarity correlates highly with program maintainability and correctness. For the purpose of ranking programs in the samples, clarity was defined by the following list of attributes:

(1) Concise (lack of verbosity in number of *source* statements)
(2) Straight forward (lack of tricky, obscure code)
(3) Understandable (good comments, formatting, use of mnemonics)
(4) Clear control structure (good use of structured programming, lack of undue complexity, modularization)
(5) Uniform style (changes well integrated; few, if any, "historical reasons" for the present state of the program)
(6) Self-contained with respect to documentation
(7) Appropriate use of macros
(8) Appropriate use of change levels, i.e., adherance to change control procedures.

B. Selection and Grading of the PW Sample

Prior to sampling, it was decided to eliminate arithmetic programs (e.g., those that contain floating point operations) from consideration and consider only control programs. It was felt that the basic differences in purpose and content among programs ostensibly representative of the same population might have been sufficient to mask otherwise significant coding differences indicative of program quality. From the 250 PW programs, over 110 remained for study purposes, and enough time was not available to study the arithmetic category. Fourteen PW programs (referred to as programs 1-14) were randomly selected. No attempt was made to "weight" programs according to size or importance; each had an equal chance of appearing in the sample. The 14 programs consisted of 4978 statements, excluding comments. Program size ranged from 34-1065 statements, with a median of 180.

Of the three graders, one person was the designer of ETC, another was an ETC programmer and the third was a highly skilled programmer who learned ETC for this experiment. Each of the programs was painstakingly graded on every criterion by the raters. Only after each of the three raters had independently completed the evaluation of the 14 programs, were results compiled and compared. In those cases where individual estimates disagreed, differences were surprisingly easily resolved by a process of joint review. Program clarity was the only criterion of quality for which a lengthy discussion was required to reach a consensus.

The argument that a random sample of programmers should do the grading is fallacious. One certainly does not want to include the judgment of a poor programmer in

riving at the consens ding. It would be desirable, however, to have an independent good programmer verify the prediction formulas by judging a second set of programs.

The question of what is the "proper" sample size for the initial set of programs is a very difficult one to answer. Traditionally in statistics [9], sample size determinations are based on the particular question which the data are intended to answer and also the "expected" variability of the data. When data are to be collected to answer many different questions, when the forms of the relationships are unclear, when little prior knowledge of the extent of variability is present, the situation is far less obvious.

The selection of 14 as the initial sample size was based more on feasibility than statistical considerations. If the prediction equations based on the sample had not been "reasonably" effective in predicting the quality of additional programs, one possible explanation would have been the small number of programs in the sample. (On the other hand, with a very strong relationship between the Ys and Xs the sample size does not have to be large.)

Basically, the initial sample could be considered as an attempt to get some measure of the variability in the data, both with respect to the range of the Xs (easy to do) and the "strength" of the relationship of the Xs to the Ys (much more difficult). The fact that so much of the variation is "explained" by relatively few independent Xs indicated that some degree of homogeneity was present in the data.

3. Possible Independent Variables

The independent variables, selected after the grading was completed, are listed in Table 1. (Suggestions from the literature, combined with judgement and intuition, formed the basis for selection.) Most of them are self-explanatory; a few, however, deserve some explanation.

For X_1, the machine operation IJNO (index-and-jump) is generally generated by the compiler if and only if a DO loop has been expressed in such a manner as to generate optimal code. Thus, (DO loops-IJNOs) is a measure of goodness of object code, assuming that all DO loops can be (re)phrased to generate IJNOs.

For X_2 and X_{17}, ".A" refers to the use of the .A qualifier (i.e., referencing only the address portion of a word); "A0" refers to the explicit use of the variable referencing the A0 register; and "ughs" refers to the occurrences of the character strings FLAG, FLG, SW, or SWITCH in the source program. It was believed that "good" programs do not use .A (such (they should use the ADDR attribute in the declaration of the variable), do not use A0 explicitly (it often indicates unnecessary machine dependence and is a dangerous practice since ETC uses A0 as a scratch register), and do not have variables whose names include the character strings FLAG, FLG, SW, or SWITCH (this is indicative of flag or switch variables, considered a bad practice in general).

For X_3, X_4, and X_5, "longest run of labels" refers to the longest contiguous run of labels in the cross reference listing. The cross reference listing produced by the compiler lists all the identifiers (labels and variable names) of a program alphabetically. Thus the longest run of labels could be considered a measure of how "bad" the label mnemonics are. Before the program had been written to extract data from the listings, it was noted that it was easier to count the number of labels in a "bad" program than in a "good" one. Why? The labels in bad programs seemed to occur in blocks in the cross reference listing. It was later noted that long runs of labels are often an attribute of programs that had been transliterated from FORTRAN to ETC, the ETC labels being created by prefixing each FORTRAN statement number by an "L."

TABLE I. Independent variables

X_1	:= (DO loops - IJNOs) / machine operations
X_2	:= (.As + A0s + ughs) / source statements
X_3	:= longest run of labels / labels
X_4	:= longest run of labels / source statements
X_5	:= longest run of labels
X_6	:= identifiers referenced / identifiers
X_7	:= flags / source statements
X_8	:= (DECLARFs + ACTs) / source statements
X_9	:= labels
X_{10}	:= machine operations / source statements
X_{11}	:= machine operation fetches / machine operations
X_{12}	:= machine operation moves / machine operations
X_{13}	:= (fetches + moves + stores) / machine operations
X_{14}	:= source statements
X_{15}	:= ACTs / source statements
X_{16}	:= DECLARFs / source statements
X_{17}	:= A0s / source statements
X_{18}	:= ughs / source statements
X_{19}	:= source level comments / source statements
X_{20}	:= simple assignment statements / source statements
X_{21}	:= DOs / source statements
X_{22}	:= (BREAKs + CALLs + DOs + ELSEs + IFs + RETURNs + ITERATEs + WHENs + DO loops + GOTOs) / source statements
X_{23}	:= SRIs / machine operations
X_{24}	:= UCIs / machine operations
X_{25}	:= BASEs
X_{26}	:= DATAs
X_{27}	:= macro definitions
X_{28}	:= (IFs + WHENs + DO loops + CALLs)
X_{29}	:= X_{28} / source statements

Programs mindlessly transliterated from one language to another are usually poorer in quality for two reasons. Firstly, the transliterated program does not take advantage of features in the new language. Secondly, the transliterated programs contain "historical" constructions (due to the old language) and thus clarity suffers.

For X_6, "identifiers referenced" refers to those identifiers actually referenced in a program. Since the programmer has the (default) option of making known to the compilation all variables in a data section (and not just those used), "identifiers referenced" is not necessarily the same as "identifiers declared." X_6 was thought to be a measure of how well a programmer specifies his input and output variables explicitly. Furthermore, clarity suffers when the cross reference listing contains many variables not referenced in the program.

For X_7, "flags" refers to the number of compiler-generated flags. These include warnings as well as fatal errors (e.g., "undefined variable"). For X_{11}, X_{12}, and X_{13}, "fetches," "moves," and "stores" are respectively the number of machine operations that are fetches from memory into registers, moves from one register to another, and stores from registers into memory. Moves from memory to memory do not exist in the machine language repertoire.

For X_{20}, "simple assignments" are assignment statements of the form "var1 = var2." For X_{21}, the "DOs" represent the number of DO groups. For X_{22}, the sum represents all types of control statements. For X_{23}, the machine operation "SRI" indicates a [illegible]

age. For X_{24}, the machine operation "UCJ" indicates an unconditional transfer of control. For X_{26}, the DATA statement indicates reference to a data section outside the am.

D. Statistical Methods Used

For the 14 sample programs, the ETC source listings and resultant object code expansions were input into a data extractor. The independent variables in Table I were constructed from the extracted data. The dependent Y variables, resulting from the hand grading, and the independent X variables, derived from the programs, were then used in a iple regression with a transformation of the dependent variables. The Appendix contains the details of the transformation chosen.

E. Independent Variables Employed

For each prediction equation yielding $Y_1, \cdots, Y_5$, various combinations of the 29 possible independent variables were extensively explored. The selection was based on the following criteria:

(1) High square of multiple correlation coefficient (R^2)
(2) Six or fewer X variables
(3) Independent variables having some justification for their inclusion other than "statistical" value
(4) Attempt to have the sign of the coefficient agree with the sign of the coefficient in the single variable regression equation.

R^2 reflects the percent of total variation "explained," the first criterion is an obvious one. The second was imposed because few variables would be expected to be common to larger sets of X variables in the larger population of interest, because only 14 programs were being used for estimation, and therefore *any* set of 13 X variables would yield $R^2 = 1$, and because of a hope that a few variables would be sufficient to yield a "high" R^2 value. The third criterion was based on a desire to be able to "physically" interpret (understand) the resulting equations and also in the belief that their predictive ability would be enhanced. It might be noted that a fairly intensive search which did not impose this criterion failed to materially improve the R^2 obtained. The fourth criterion makes explicit a portion of the third and is used for the same reasons.

Three variables (X_8, X_{10}, X_{21}) occurred in three out of five equations. Three (X_1, X_{20}, X_{22}) occurred twice. Eleven variables (X_2, X_3, X_5, X_9, X_{11}, X_{15}, X_{16}, X_{17}, X_{18}, X_{23}, X) occurred once.

The Appendix contains the five final prediction equations for Z, a transformation of Y, including the Students "t" value (a measure of the importance of each variable) for each coefficient, and examples of the differences between the observed and predicted values.

One comment is g the equations selected seems in order. The equation for Z_4

has only four X variables and the lowest R^2. It might be wondered why more X variables were not selected to yield a better prediction. An extensive search failed to find a better set. For example, the best set of eight variables found, including these four, had an R^2 of only 0.85. Either the X variables used were not the "right" ones for predicting Z_4; more "powerful" methods were needed; Z_4 is simply more difficult to predict than the other measures of efficiency; the particular 14 programs were not representative of the population; or the ratings given to Y_4 did not correspond closely enough to the "true" percent of source statements which can be improved with difficulty.

F. Validation of the Formulas for PW

To validate the prediction formulas for the PW population, it was decided to compare the predicted and hand-graded values of an additional sample. The selection and grading were done as described in Section B. Unfortunately, time allowed only four programs to be selected (programs 15-18); only two raters were available (including one not included among the original raters); and only three qualities (Y_1, Y_3, and Y_5) were considered. Table II compares the predicted and actual values for the four additional programs. The detailed discussion of the statistical validation found in the Appendix presents results that confirm the usefulness of the prediction equations.

TABLE II. Predicted vs Observed Y values for programs 15–18.

	PROGRAM			
	15	16	17	18
Y_1	6	4	16	16
$\hat{Y}_1$	0.3	3.3	10.0	10.4
p for Y_1	0.09	0.86	0.34	0.43
Y_3	29	29	26	43
$\hat{Y}_3$	35.3	40.5	14.7	44.8
p for Y_3	0.34	0.16	0.01	0.83
Y_5	4.87	5.25	10.5	9.88
$\hat{Y}_5$	9.72	7.62	5.17	16.89
p for Y_5	0.21	0.59	0.14	0.30

G. Validations of Formulas for MW

Having tentatively validated the prediction equations for the PW population, our attention was focused on the MW population and the possibility of applying the formulas to that set of programs. It was not known if such differences as personnel, programming techniques and conventions, functional capabilities of the systems, and sizes (object and source) would be sufficient to bias the formula's predictability. The selection, grading and comparison approach used for PW (programs 15-18) was not used because of time constraints; instead, a "backward prediction" scheme was employed to validate the equations. Acting on a suggestion of J. W. Tukey, the five predicted Z values for each of the original 14 programs were used to try to estimate some of the X variables.

The general result of this investigation (the details of which are given in "

Appendix) is that, e ough the R^2 values show that the $\hat{Z}$ values do not predict the Zs very well, additio rograms from both the PW and MW sets are predicted about as well as the original 14 (using the equations generated by the original 14 programs).

Although it appears very difficult to quantify the statements made above, it nevertheless appears reassuring that comparable goodness of predictions of the X variables are possible for the MW and PW sets. This lends additional evidence to the results of the (PW) analysis (Section III. D.), in which the two sets of data are very comparable.

III. ANALYSIS AND APPLICATION OF THE PREDICTION FORMULAS

A. Independent vs Dependent Variable Correlations

Table III presents the simple (linear) correlations R, between each X variable and each Z variable for programs 1–14. The sign indicates either a negative or positive relationship

TABLE III. Independent vs dependent variable correlation matrix.

X-VARIABLE	Z_1	Z_2	Z_3	Z_4	Z_5
1	0.57	0.60	-0.19	0.34	0.28
2	-0.07	0.10	-0.11	-0.29	0.18
3	0.29	0.38	0.58	0.33	0.29
4	0.27	0.33	0.43	0.22	0.28
5	0.22	0.41	0.37	0.29	0.47
6	-0.08	-0.04	0.67	0.21	0.17
7	0.09	-0.20	-0.32	-0.27	-0.33
8	-0.32	-0.39	0.29	-0.08	-0.33
9	0.11	0.33	-0.08	0.34	0.55
10	-0.37	-0.40	-0.26	-0.40	-0.44
11	-0.46	-0.07	0.21	-0.01	0.14
12	0.48	0.37	-0.06	-0.002	-0.25
13	0.03	0.29	0.06	0.16	0.05
14	0.10	0.28	-0.06	0.35	0.50
15	-0.54	-0.44	0.27	-0.06	-0.09
16	-0.15	-0.30	0.26	-0.08	-0.43
17	-0.08	-0.26	-0.07	-0.40	-0.21
18	-0.03	0.10	-0.12	-0.23	0.02
19	0.11	-0.12	0.15	-0.21	-0.41
20	0.21	0.21	0.50	0.37	0.12
21	0.60	0.55	0.19	0.61	0.53
22	0.72	0.73	0.17	0.43	0.55
23	-0.21	-0.39	-0.50	-0.31	-0.19
24	0.19	-0.03	0.58	0.27	0.05
25	-0.09	0.22	-0.53	-0.04	0.35
26	0.18	0.49	-0.14	0.25	0.65
27	-0.02	0.19	-0.46	0.18	0.42
28	0.08	0.25	-0.11	0.38	0.48
29	0.63	0.61	-0.14	0.36	0.47

between a given pair of variables; absolute values greater than 0.46 are signifi different from zero (5% level). Note that a high individual predictive value d guarantee inclusion of that variable into the prediction equation. (See Section II.E. of the Appendix.) It may very well be that, since several X variables are included, a combination is more predictive than an individual X.

Complexity, as measured by proportion of control statements (X_{33}), seems to be a good discriminator of program goodness for almost every criterion (R = 0.72, 0.73, 0.17, 0.43, 0.55 for $Z_1, \cdots, Z_5$). In other words, bad programs are more complex, not just more verbose, than they need be.

The more DATA statements (X_{36}) in a program, the worse the clarity becomes (R = 0.65). That is, the less localized the data references, the worse the clarity. The use of DO groups (X_{21}) is relatively highly positively correlated with all of the dependent variables (except Y_3). Does this mean that the use of DO groups is "bad"? Probably not. A particular instance of the DO groups, however, was frequent in programs (among 1–14) of bad quality. This took the form

```
· · · THEN DO
simple statement
DOEND
```

instead of just

```
· · · THEN simple statement.
```

That is, a one statement DO group was used where none was needed.

Z_4, the extent to which reductions in the number of source statements can be made via complete restructuring, appears to be the most difficult dependent variable to predict by a single X variable. It might also be noted that, in most cases, the sign of the relationship between an X variable and each Z variable is the same. (X_{35} and X_{27} are striking counterexamples.)

B. Independent Variable Intercorrelations

One of the more interesting inferences from the simple correlations among the independent variables is that labels are useful even if the program is structured. This may be derived by first noting that there is no correlation between the number of source statements in a program and the quality of structured programming (measured by GOTOs normalized by source statements). Since there is, however, a positive correlation of 0.95 between number of source statements (X_{14}) and number of labels, there can be no correlation between number of labels and use of structured programming.

That programs do not get more complex as they get larger is indicated by the correlation of 0.97 between the number of source statements (X_{14}) and the number of control statements (X_{33}), taking control statements divided by total number of statements as a measure of complexity. An R of 0.50 between program size and clarity indicates that clarity does suffer as programs get larger.

C. Dependent Variable Intercorrelations

C leration of Table IV, the dependent variable intercorrelation matrix (in terms of Z_i), leads one to draw several conclusions. From comparisons of correlations of Z_4

TABLE IV. Dependent variable intercorrelation matrix.

	Z_1	Z_2	Z_3	Z_4
Z_1	0.84	0.17	0.53	0.46
Z_2		0.11	0.58	0.74
Z_3			0.44	0.22
Z_4				0.70

Z_2 with Z_3 and of Z_1 and Z_2 with Z_4, we see that the reductions of source due to complete restructuring of programs are more likely to reduce object code size than are reductions of source due to mechanical transformations. From the correlations between the sets (Z_1, Z_2, Z_3, Z_4) and Z_5, we see that the clearer a program is, the more difficult it is to make reductions in source or object via a complete restructuring. Conversely, mechanical transformations of a program are less likely to improve clarity of a program than are basic restructurings.

While these findings are perhaps not too surprising, they do tend to confirm the validity of our quality criteria, especially to the extent to which the correlations seem "obvious." This is true because such correlations were *not consciously considered* by the pr ammers making the quality rankings.

D. Structured Programming and Clarity

As a sidelight to the main purpose of the experiment, some analyses were made of the relationship between structured programming and clarity. Many of the programming groups were required to code using only the structured programming features of ETC (e.g., WHEN, ELSE, CASE, but not GOTO, BREAK, ITERATE, IF). Insofar as the use of GOTO indicates a lack of structured programming, one would expect a correlation between the use of GOTOs (either conditional or unconditional) in a program and a program's clarity ranking. Indeed, it was hoped that GOTOs would be a good independent variable to correlate with clarity.

Figure 1 is a plot of the number of GOTOs against the predicted clarity ranking Z_5 for all PW programs with at least 20 GOTOs. From a superficial examination, it appears clear that there is little, if any, relationship on an overall basis. However, there is some indicat that clarity is related to programs with greater than forty GOTOs. The (at least tentative) conclusion indicated by the detailed analysis in the Appendix is that poorer clarity is associated with programs of more than forty GOTOs. Of course, it be recognized that these tentative conclusions depend on at least two critical assumptions: the correspondence of the estimated Y_5 with "clarity," and the identification of fewer GOTOs with "more" structured programming. The latter assumes the nonrelevance of a measur program size (such as number of source statements) with number of GOTOs. elow.) A plot of the number of GOTOs against the predicted clarity ranking Y_5 for the MW programs is strikingly similar to Figure 1.

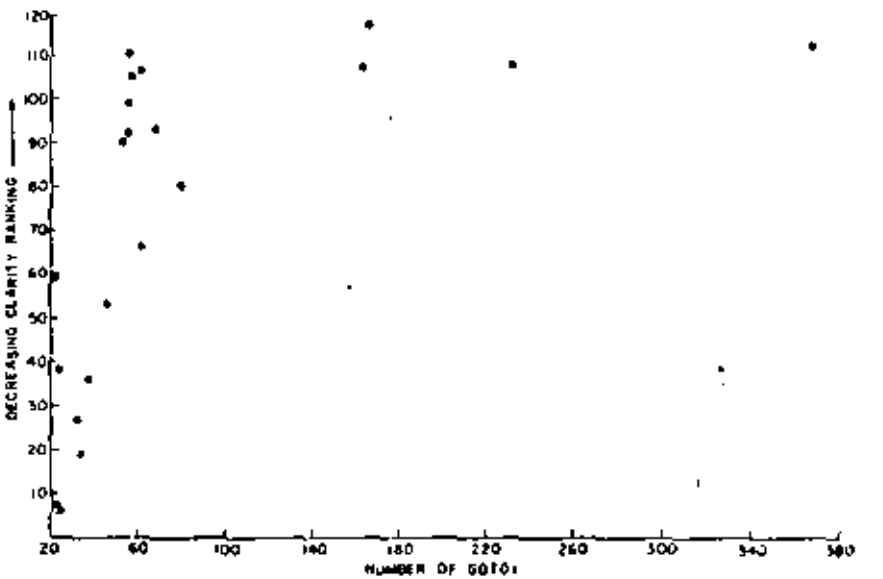


Fig. 1. Clarity ranking vs GOTOs (PW)

E. Comparisons of Functional Groups

The two systems studied were divided into the functional areas of hardware and software test functions, interlocations communications, CRT display subsystem, software coordinator, real-time I/O, and compute-bound transaction module. One would assume that to provide minimal system capabilities, some of these functions are more critical than others. To the extent that our quality criteria are related to system reliability, one would hope that the critical functions are of better quality. To test this hypothesis, five groups of PW programs were ranked on an overall basis with respect to the predicted $Y_1, \cdots, Y_5$ values. The PTP group has a noticeably larger average rank (worse programs), and the PWT group a slightly lower rank (better programs). One group was noteworthy in being the worst group with respect to Y_3 and Y_2 and the best with respect to Y_3 and Y_4. The small difference in average ranks on the one hand between Y_3 and Y_2, and on the other hand between Y_3 and Y_4, is an indication of the consistency of the data.

The results of the analysis described in the Appendix are what we would hope: PWT is better than PTP. PTP is a hardware test function – the system can function in a limited mode with low quality code here. PWT is a critical real-time processing function – high quality code would be desirable.

F. Real-time Priority and Quality

Both PW and MW are priority-scheduled systems, that is, with each task (or program) is an associated priority. The particular task initiated by the schedul highest priority task ready to execute. The priority selection ground rules for systems included:

(1) High priority must be given to tasks with critical response or completion times
(2) Low priority may be given to tasks whose execution is not essential to the success of the mission, e.g., TTY output, spooling programs
(3) Intermediate priorities should be assigned after considering criticality of function, response time, overload, etc.

A partial plot of real-time priority numbers vs quality for the major programs comprising PW tasks is presented in Figure 2. The lower priority numbers correspond to higher

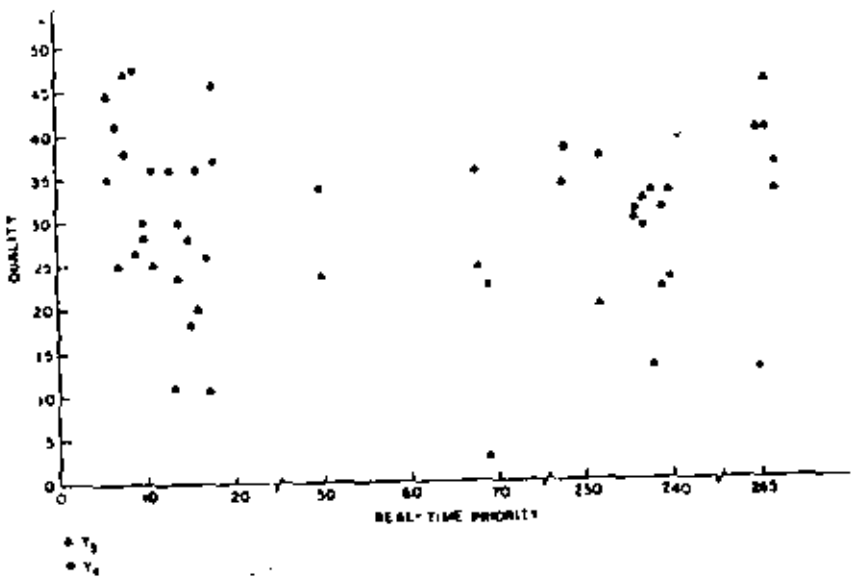


Fig. 2. Real-time priority vs quality.

priorities. Casual inspection indicates no correlation between priority and quality. Could it be that priority is not known during the coding stages? Unfortunately, time did not permit further analysis of the implications presented in this plot.

G. Program Versions and Quality

As most large scale systems are made available for general use in a series of incremental releases, we were interested in determining whether later versions were "better" or "worse" than earlier versions. Due to time limitations, we attempted this determination on a very small scale; for 12 of the 14 programs in the original sample (PW Version 3) we were able to get versions included in a later software release (Version 4). The prediction equations were run against each of the "original" PW programs except 7 and 12. A plot of the differences (PW Version 4 minus PW Version 3) for each program-variable combination indicated that some are better; others worse; some haven't changed much; and for two programs the decision is unclear. Apparently, the answer as to whether software quality has improved or not depends on which program one has in mind.

One would expect minor degradations in clarity and conciseness in programs that were "patched": as the word implies, such changes are not well integrated into the structure of the program. Along these same lines one might infer that two programs were rewritten substantially, and the job was botched for one of them.* Unfortunately, we did not have time to verify these inferences.

IV. CONCLUSIONS

A. Summary of Results

The results from this study were most encouraging and should lead to additional study in this field. We have derived formulas that mimic the judgement of good programmers about the quality of programs. Thus, with the aid of the tools developed in the study, program managers are now able to identify, within the PW and MW collections of control programs, likely candidates for quality improvement. While the formulas are not directly applicable to other languages and other program populations, the methods used in this study may be used to derive analagous prediction formulas. The tools have also been effectively employed to compare programs within a functional group to those within another functional group. Functional areas to be improved may thus be identified.

Some further light has been shed upon the question of clarity versus structured programming. Our findings tentatively indicate that a large number of GOTOs (absolute number or ratio to number of source statements) implies low quality ratings. The converse was not true, however. Other correlations among the variables indicated that bad programs are more complex, not just more verbose, than they need be, and the less localized the data references, the worse the clarity.

While programs do not get more complex as they get larger, clarity does suffer as programs get larger. Also, reductions of source due to complete restructuring of programs are more likely to reduce object code size and improve clarity than are reductions of source due to mechanical transformations. While some of these statements may not be too surprising, it should be realized that the statements are *results* and not *a priori* assumptions.

B. Comparison with Other Studies

Weinberg [3] has reported the findings of a study comparing the qualities of program core size, program clarity, output clarity, number of statements, and hours to produce a program, to the objective of the programming group. He seems to have given one program specification to several programming teams, each with a different primary objective. We are not told how the clarity rankings are derived. One subjective conclusion that I draws (at least no data are given to support the conclusion) is that "achieving the ... objectives of minimal statements and minimal core are related in an obvious ...

achieving the objectives of minimal statements tends to be correlated to some extent with *program* clarity–which our raters attributed to a shorter program being simpler to read, all other things being equal." Weinberg does supply data to show that each team ranks first on its primary objective. If we assume, for the purposes of our study that primary objective and quality rankings are highly correlated, then using our measures of quality, we should be able to make some comparisons with the Weinberg study. Indeed, as Table IV shows, there is some correlation between minimal core (Y_3) and minimal statements (Y_4), although the correlation is not very strong. (Only Y_2 and Y_4 can really be used as measures of programming objectives; it can be argued that Y_1 and Y_3 do not represent basic objectives.) Also, from Table IV, minimal statements (Y_4) shows some correlation with clarity (Y_5).

Akiyama [1] uses simple regression analyses to estimate the number of bugs based on the "nature of the program." By "nature of the program" he means "the sum of the numbers of decision symbols and subroutine call symbols appearing in the flow chart made for each module." The high correlation coefficient of 0.92 is due perhaps to the homogeneity of his programmer population (and possibly the programs themselves) and the apparent cleanliness of his bug data.

Our study yielded a correlation coefficient of 0.56 between (IFs+CALLs+WHENs+DO loops)$^{\frac{1}{2}}$ (source statements and clarity ranking), indicating some correlation of clarity with the nature of a program. If one assumes a correlation between bugs and clarity, then we tend to confirm Akiyama's conclusion that there is a correlation between bugs and the nature of a program. Akiyama gets a correlation coefficient of 0.82 for the relationship between program steps and bugs, while we get a coefficient of 0.25 relating number of source statements to clarity. Thus, either clarity is not highly related to bugs, or the measures of program steps are not analogous.

C. Suggestions for Future Study

One eventual goal of such studies is to derive an equation predicting the bugs/time/lines of code for a program. To do this it is necessary to have a reliable bug history (trouble report history) for a sample of programs. This was not available for the present study, but with such histories, the method given in this study could be so extended. It would be very desirable to explore the relationship between quality and bug history.

A cost function might be derived (using the quality analysis equations derived in this study) to suggest where reprogramming effort should be invested. Suggestions for improvement of programs might be provided automatically. Listing the independent variables and their coefficients in each equation would be a start. Indicating which variable actually contributed most to determining the quality assessment would be helpful only if the variable were a *direct* measurement of the quality of a program.

In an excellent introductory article in [11], Grenander and Tsao mention the surprisingly few examples of the use of regression analysis in computer performance evaluation. Although the emphasis of this paper is different from their theme, there are several important points which need further exploration. For example, some of the system evaluation tools discussed in [11] could be used to provide feedback data to the prediction method described in our paper.

As an aid to the programmer wishing to focus on a particular section of a program for quality i[illegible]vement, quality gradings might be made on a subroutine or BEGIN

block basis, rather than a program (i.e., unit of compilation) basis. This would also improve the accuracy of the grades in those cases where a program was written by sev programmers. To increase confidence in the present set of equations, the set of prese equations might be compared against a new set determined from the presently-predicti very "bad" and "good" programs. Since arithmetic programs were excluded from the original sample, the extent to which the study results apply to such programs might be investigated.

Several additional statistical analyses might be performed. Since the variance of $\hat{Y}$ is still a function of N, a weighted regression analysis might be a useful alternative to t presented here. To increase the confidence of the consensus grading, differences amo graders should be evaluated. A technique of Carroll [2] could be used. Other indepe dent variables, combinations of them, and transformations might be considered. A experiment might be devised to determine whether a cause and effect relationship ex between the "good" X prediction variables and quality. The relationship between structured programming and quality should be investigated more fully.

APPENDIX

DETAILED DISCUSSION OF STATISTICAL PROCEDURES

(Sections in this appendix consist of detailed statistical discussions of identically numbered sections in the body of the paper.)

II. D. Statistical Methods Used

Although the Y_i ($i = 1$–4) were immediately interpretable values, as the propor a program easily improved, etc., their use as dependent variables in a regression was advisable for two main reasons. Firstly, if it were assumed that

$$Y_{ij} = \sum_{k=1}^{r} \beta_{ik}^{*}(r) X_{jk} + \epsilon_{ij}^{*} .$$

where Y_{ij} is the observed Y_i value for the jth program and $\beta_{ik}^{*}(r)$ is the (i,k) regressi coefficient when r variables are considered (actually the β value depends on the par r variables); X_{jk} is the observed value of the kth independent variable for the jth pr ϵ_{ij}^{*} is the "error" (deviation from the model); then for fixed βs, there would be poss values such that the predicted Y would lie outside of 0 and 1. Secondly, one assum for ordinary linear regression is that the variance of Y_{ij} is a constant. Since Y_{ij} is a proportion, this is clearly not satisfied

$$\mathrm{Var}(p) = p(1-p)/N .$$

A transform [10] meeting both of these objections is

$$Z_{ij} = 2 \sin^{-1} (\sqrt{Y_{ij}}). \quad (2)$$

There was some preliminary investigation of the use of (1) which indicated that the transformation (2) would produce "better" results in the sense of a higher multiple correlation with the same variables. The square of the multiple correlation coefficient R^2, is the proportion of the total variation "explained" by the particular X variables used in the multiple regression. Table I defines the X variables which were originally considered for use in this study.

A combination of step-up and step-down regression procedures (adding or eliminating one X variable at a time) were used to obtain some "good" estimating equations for each of the Z_i's ($i = 1, 4$). The criterion used for "goodness" was the multiple correlation obtained. Since there were many more variables than observations, conventional means for implementing step-down regression were not appropriate. What was actually done was to start with a set of variables which might (from analysis) be expected to influence the particular response, and to step-down from there. This might be done with several different sets of starting variables. One criterion used was that the variables common to several "final" sets would be the ones chosen.

As the 14 observations were only a small sample from the population of interest, it was not desirable to use enough variables to make R^2 close to unity. Rather, an attempt was made to find a "small" set which yielded a "large" R^2. The idea was simply that it would be expected that a smaller set of variables (common to several sets) would be more likely to be good predictors in the larger population of interest.

A brief investigation was made of the applicability of discriminant analysis to this problem, but the small size of the sample and the lack of resources to assign the "good" or "bad" label to many programs, led to a negative conclusion. In the case of Y_5 (a program's clarity rank) no obvious transformation seemed appropriate, so it was decided to see how good a prediction, with respect to a ranking of the initial fourteen programs, would be possible. It was expected that for some of the total population "impossible" rankings (outside of $1, \cdots, 14$) would result, but this itself might be indicative of a very good or bad program.

II. E. Final Prediction Equations

In the following, t is the observed value of a variable having a "Students" t distribution. "Large" values indicate that the coefficient significantly differs from 0. X_0 is identically 1; that is, the coefficient associated with X_0 is the constant term in the equation. For example

$$\hat{Z}_1 = 0.96 + 2.63X_1 + 0.42X_3 - 0.19X_{10} \cdots .$$

Variable Z_1, $R^2 = 0.90$

X VARIABLE	COEFFICIENT	t
0	0.96	5.50
1	2.63	1.32
3	0.42	2.85
10	-0.19	-2.35
11	-1.55	-2.47
15	-2.91	-3.13
21	1.30	1.98

Variable Z_3, $R^2 = 0.91$

X VARIABLE	COEFFICIENT	t
0	0.53	2.17
1	7.96	3.62
5	0.07	3.49
6	-0.67	-1.93
10	-0.12	-1.25
22	1.62	4.09

Variable Z_4, $R^2 = 0.93$

X VARIABLE	COEFFICIENT	t
0	0.25	2.80
2	1.84	4.08
6	1.09	7.62
16	2.84	6.53
21	1.80	4.18
24	0.76	1.84

Variable Z_5, $R^2 = 0.77$

X VARIABLE	COEFFICIENT	t
0	1.20	7.27
10	-0.11	-1.46
17	-2.52	-2.94
20	0.48	1.80
21	2.14	3.60

Variable Z_1, R^2 = 0.86

X VARIABLE	COEFFICIENT	t
0	3.45	1.17
6	11.57	1.83
9	0.07	4.28
18	-95.90	-2.81
20	-27.45	-2.95
22	36.49	4.33
21	-54.46	-2.19

PROGRAM	OBSERVED VALUE (Z_i)	ESTIMATE ($\hat{Z}_i$)	RESIDUAL ($Z_i - \hat{Z}_i$)
1	0.643	0.609	0.034
2	0.495	0.524	-0.029
3	0.000	0.030	-0.030
4	0.573	0.554	0.019
5	0.976	0.989	-0.012
6	0.676	0.631	0.045
7	0.348	0.363	-0.015
8	0.451	0.607	-0.156
9	0.644	0.499	0.145
10	0.707	0.702	0.005
11	0.609	0.667	-0.058
12	0.535	0.512	0.023
13	0.609	0.545	0.064
14	0.403	0.438	-0.035

II. F. Validation of the Formulas for PW

To validate the prediction formulas for the PW population, the predicted and hand-graded values of an additional sample were compared. To obtain a numerical measure for the "goodness of fit," a "t" statistic was calculated for each additional program, and the corresponding probability p, of observing a difference as large or larger was obtained. The values in Table II give the "p" value for each program, Y combination. For example, the probability is approximately 0.09 that a difference as large or larger than the observed difference between Y_1 and $\hat{Y}_1$ (predicted Y_1) could have arisen by chance if in fact they were drawn from the same population. Low probabilities indicate "poor" prediction.

The question of what p value is "significant" is a difficult one to resolve. Conventionally, statisticians have used 0.01 or 0.05 as the critical level. The smaller the level taken, the fewer errors of the first kind which can be made (i.e., saying that there is a difference when none exists). However, the smaller the level, the larger is the probability of making an error of the second kind (i.e., saying that no difference exists when one is present). A formal analysis would take into account the "loss" incurred when errors of each kind were made.

To combine the p values into a single overall measure, the (false) assumption was made that the "t" statistics were independent (not true because of the common estimate of σ^2). With this assumption, Fisher's method [5] for combining probabilities could be applied. This consists of calculating

$$C = -2 \sum_{1}^{K} \operatorname{Ln} p_i \ ,$$

and comparing this value to a chi-square distribution with $2K$ degrees of freedom. The C_i values ($i = 1, 3, 5$) corresponding to Y_1, Y_3, and Y_5 are

$$C_1 = 8.85$$
$$C_3 = 15.41$$
$$C_5 = 10.53 \ .$$

Since the critical value, at a 10% level of significance, is 13.4, only C_3 can be considered as being indicative of a poor fit. The overall C of 34.79 is barely larger than the 10% critical value of 33.2. It might be noted that one single prediction is responsible for 25% of the total C. The difference for Y_3, program 17, contributes 9.21 to the total. Eliminating this yields

$$C'_3 = 6.21, \text{ and}$$
$$C' = 25.58 \ .$$

Since the 20% significance level for C' is 27.3, the 11 predictions are not significantly different from the actual measurement. Stated differently, it appears that 11/12 of the predictions are "good" in an overall sense. However, it is not at all obvious what might be "peculiar" in the one instance.

II. G. Validations of Formulas for MW

Acting on a suggestion of Tukey, the five predicted Z values for each of the original 14 programs were used to try to estimate some of the X variables. For example, for say X_1, a regression equation of the form

$$X_{1k} = b^{(0)}_{(1)} + \sum_{i=1}^{5} b^{(i)}_{(1)} \bar{Z}_{ik} \ ,$$

was determined, where $k = 1, \cdots, 14$. Using the derived $b^{(i)}_{(1)}$, estimates of X_{14} ($\hat{X}_{14}$) could be calculated. Squared correlations from 0.16 (for X_{10}) to 0.84 (for X_{21}) were obtained, indicating wide variability in the ability to predict various X variables. Surprisingly, there is no significant difference in the predictive ability of those X

included or not incl ... the Z regression equations. A nonparametric rank sum test yielded a nonsignific... fference (25%).

In most cases there is little difference between the ability of the equations (calculated from the Ys for programs 1-14) to predict Xs for the original set of 14 or an additional set of four which was selected. In most cases the "extreme" program appears to be program 18. Since this program is an extremely large one, having 2611 source statements, this may at least partially explain the poor fit. (The program is the largest in PW, one-half again larger than the next largest program.)

The results of nine programs (19-27) from the MW set, for selected Xs predicted by the same prediction equation as the original 14 (from the PW set), were examined. Except for X_{16}, which appears to be a "poor" fit, the equations seem to predict Xs in this set about as well as those in the original set.

III. D. Structured Programming and Clarity

To obtain a numerical measure of the relationship between GOTOs and clarity for the PW data, the rank sum statistic (see [6]) T, was evaluated for those X values greater than 40 with the following result:

CATEGORY	NUMBER	AVERAGE RANK
$X <= 40$	99	51
$X > 40$	13	94

As might be expected from the large difference in average ranks, the corresponding normal variable

$$n = (1221-131 \cdot 112/2)/\text{sqrt}(13 \cdot 99 \cdot (112/12)) = 4.9 ,$$

is very significant. The conclusion from this set of data is therefore, that programs with X greater than 40 come from a population having less clarity than those with X less than or equal to 40. That is, structured programming does seem to be associated with improved clarity.

To evaluate the extent to which clarity is related to the number of GOTOs greater than 40, a rank correlation r, was calculated

$$r = 1-(6 \cdot 192)/(13 \cdot 168) = 0.47$$

A comparison of this value with Table 10 in Conover [6] shows that this correlation is just significantly different from 0 at the 5% level. The (at least tentative) conclusion is therefore that poorer clarity is associated with programs of more than 40 GOTOs.

A plot of the number of GOTOs against the predicted clarity ranking $\hat{Y}_5$ for the MW programs revealed a striking similarity with Figure 1. Based upon the PW analysis, a cutoff number of 40 was chosen for the number of GOTOs. The rank sum statistic T was therefore evaluated for those X values greater than 40, with the following result:

CATEGORY	NUMBER	AVERAGE RANK
$X <= 40$	198	110
$X > 40$	76	208

Interestingly, the ratio between the average ranks (1.9) is almost the same as the ratio for the PW programs (1.8)!

For the MW data, the corresponding normal variable

$$n = (15844-76 \cdot (274/2))/\text{sqrt}(76 \cdot 198 \cdot (274/12)) = 7.6 ,$$

is also significant. Confirming the conclusion reached for the PW data, the MW programs indicate that those with more than 40 GOTOs come from a population with less clarity than those with fewer. The rank correlation

$$r = 1 - (6 \cdot 29410/(76 \cdot 76 - 1)) = 0.60$$

is also highly significantly different from 0.

The agreement in the results from the two sets of data is very comforting. On a proportional basis, the MW programs had three times as many programs with more than 40 GOTOs as the PW set. To try to measure the effect of the number of GOTOs on predicted clarity, the $\hat{Y}_5$ values were ranked, i.e., the smallest (best) $\hat{Y}_5$ was assigned the rank 1, the next largest 2, etc. Each of the observations in each set (say in the PW set) was given a unique rank (1-112). This technique assumes that there is a distinguishable difference between, say, 1.182827 and 1.19971 (programs PTRTSEAX and PMPERLOG respectively), and the former is "better" than the latter. Clearly this assumption is not true in light of the uncertainty associated with predicting "clarity."

To evaluate the effect of this assumption, alternative methods of ranking were used. However, even where it was assumed that only rounded integers were distinguishable, there was essentially no difference in the results of the analyses. Technically, the results were "robust" to the method of ranking. The assumption that, with only three programs (say), with $\hat{Y}_5$ of 2, 4, 5, they are in fact in the order 1, 2, 3 (i.e., 2 is better than 4 is better than 5) is therefore sufficient to yield the conclusions of the analysis. The $\hat{Y}_5$ were ranked before the analysis was made to obtain a known distribution. The rank sum statistic T has a known distribution [7] regardless of the underlying values, assuming only that the values with different ranks are "distinguishable." As shown above, this criterion appears to be satisfied for these data sets.

A plot of the number of GOTOs vs the actual predicted $\hat{Y}_5$ gives an impression similar to that obtained from Figure 1. An additional study (for the MW set) normalized the number of GOTOs by the number of source statements (as a measure of program size). A plot indicated that there appears to be at least as strong a relationship of $\hat{Y}_5$ ... normalized GOTOs as unnormalized. To quantify this, two rank sum tests, based on size ... greater than 0.10 and 0.20 were performed with corresponding norm... variable

$$n_1 = (22761-124 \cdot (274/2))/\text{sqrt}(124 \cdot 150 \cdot (274/12)) = ...$$

and

$$r_2 = (10248 - 47 \cdot (274/2)) \text{ sqrt}(47 \cdot 227 \cdot 274/121) = 7.7 .$$

which are both very significantly different from 0, and quite comparable to the value obtained (17.6) in the unnormalized case and to each other. The question of which analysis is more appropriate (relevant?) does not appear to be a simple one to answer.

CATEGORY	NUMBER	AVERAGE RANK
Ratio <= 0.10	150	99
Ratio > 0.10	124	183
Ratio <= 0.20	227	121
Ratio > 0.20	47	218

III. E. Comparisons of Functional Groups

To obtain an indication of the homogeneity of the different groups, the "distances" between all pairs of programs within a group were obtained, based upon the predicted Y values, by the following formula

$$d_{ij}^2 = \sum_{k=1}^{5} S_k^{-2} (Y_{ik} - Y_{jk})^2 .$$

Although the data are not presented here, there appear to be two or three programs that are noticeably apart from all others in each group.

To evaluate the statistical significance of the difference observed in the average rankings, a Kruskal-Wallis I statistic was calculated for each measure of quality. This is a nonparametric analogue of the analysis of variance [8]. The 5% critical value (with five groups) is 9.5, which indicates that the mean ranks corresponding to each quality measure except clarity exhibit an overall significant difference. Apparently, the PWT group is "better" than the others with respect to Y_1 and Y_2, and the PTP is worse with respect to Y_1 and Y_4. One quantitative measure of this is obtainable by deleting the deviant group and recomputing I based on the revised ranks without these groups. These revised average ranks indicated, since the 5% critical I in this case is 7.8, with the possible exception of the Y_4 ranks, that the remaining groups in each case can be considered homogeneous.

These results are what we would hope. PTP is a hardware test function – the system can function in a limited mode with low quality code here. PWT is a critical real-time processing function. High quality code would be desirable.

ACKNOWLEDGEMENTS

The study took a total of about three person-years, including the efforts of V. L. Bishop who hand graded programs 15–18, R. S. Roehrig who programmed much of the data analysis, and Mrs. R. Wax who performed plotting and other necessary tasks. F. A. Varrichio was responsible for the syntax specification of the ETC listing.

REFERENCES

[1] F. Akiyama, "An Example of Software System Debugging," *Information Processing*, 71 353–59 (1972).

[2] J. D. Carroll, *Multidimensional Scaling*, I, 105–55 (Seminar Press).

[3] G. M. Weinberg, "The Psychology of Improved Programming Performance," *Datamation*, 82–85 (November 1972).

[4] B. N. Dickman, "ETC, an Extendible Macro-based Compiler," *Proceedings of the Spring Joint Computer Conference*, 529–38 (1971).

[5] W. A. Wallis, "Compounding Probabilities from Independent Significance Tests," *Econometrica*, 10 (1942).

[6] W. J. Conover, *Practical Nonparametric Statistics*, 390 (New York: John Wiley and Sons).

[7] ibid., 223.

[8] ibid., 257.

[9] W. Dixon and F. Massey, *Introduction to Statistical Analysis*, 250 (New York: McGraw-Hill).

[10] C. Bennett and N. Franklin, *Statistical Analysis in Chemistry and the Chemical Industry*, 356.

[11] U. Grenander and R. F. Tsao, "Quantitative Methods for Evaluating Computer System Performance: a Review and Proposals," *Statistical Computer Performance Evaluation* (W. Freiberger, Ed.).

[12] SAFEGUARD Special Supplement, *BSTJ* (May–June 1978).

DATA TYPES AND PROGRAMMING RELIABILITY: SOME PRELIMINARY EVIDENCE

J. D. Gannon

Department of Computer Science,
University of Maryland, College Park, MD

The goal of reliable programming is to minimize the number of errors in completed progr language in which programs are written can have a significant impact on the reliability of the programming process.

One common language feature that appears in different forms in many programming lan the data type. Data types may be associated with operands in one of three ways: statically declaration; dynamically through assignment of a value; or not at all by treating operands as tions of bits. Each of these methods has adherents among the designers of programming langu However, little is known about the effect of these features on the way in which people write programs.

Some evidence concerning the effect of the various methods of associating a type with an can be gained by examining the results of an experiment which dealt with other language featu This evidence indicates that the use of dynamically typed operands results in errors which rem programs longer than errors attributable to the use of statically typed operands. Furthermore, programmers trying to convert a stupeband from one type to another, who were forced to manipula the representation of the operand, committed errors that cast doubt upon the ability of progra to write reliably in a language which treats its operands as collections of bits.

This preliminary data suggests that a more detailed and controlled experiment will be requi enable us to draw firm conclusions about the role of data types in reliable programming. A pro for such an experiment is described in the paper.

1. INTRODUCTION

Many approaches are being tried to increase software reliability: verification tha programs are consistent with their specifications; development of sets of test cases; an provision of redundant program modules. Each of these methods relies on the p mi that nearly-correct programs can be produced readily.

The goal of reliable programming is to minimize the number of errors in compleu programs. Attaining this goal may involve reducing the number of errors committed programmers and/or increasing the fraction of errors that are detected and corrected before the production of a final program. Appropriate language design can contribute both of these goals.

Presented at the Symposium on Computer Softwa
Polytechnic Institute of New York,

II. DATA TYPES

A data type partitions operands into classes in which the operands are treated as primitives by operators [5]. Data types offer two principal advantages to a programmer: abstraction and authentication [8]. The abstractions (Morris's secrecy) offered by data types allow a programmer to think in terms of his application rather than of the characteristics of the machine on which his application will run. Thus, the programmer deals with characters rather than bit patterns within a word and with matrices rather than storage structures of words. In this way, abstraction both aids the programmer in reaching a solution and prevents the programmer from writing code that depends upon a particular representation of his data, thereby precluding redesign of the program. Languages such as CLU [7] and ALPHARD [10] are being designed to offer programmers the benefits of abstraction by limiting the availability of information about the representation of data. Authentication (i.e., type checking) prevents a programmer from attempting operations on operands which are not valid representatons of the operands he wishes to process (e.g., addition of Boolean values).

In a programming language, data types may be associated with operands in one of three ways: statically; dynamically; or not at all. In a statically typed language (e.g., PASCAL), a data type is associated with a variable in a declaration. During its lifetime, the variable may only be assigned values of the type with which it was declared. In a language like GEDANKEN or SNOBOL, data types are associated with a variable dynamically during execution. The type of variable is the type of the last value assigned to it. BCPL and BLISS are called "typeless" languages. As is the case with machine code, each operand is considered to be a collection of bits (usually a word in memory). When it is applied to operands, an operator assumes that the bits represent a value of the type that the operator may manipulate. Many languages do not fit neatly into one of the three categories. For example, ALGOL 60 and PL/1 are primarily statically typed languages. However, ALGOL 60 does not require complete specification of the arguments of procedures and the UNSPEC operation of PL/1 allows the type checking mechanism to be subverted.

The type attribute in the declarations of statically typed languages is an effective form of redundancy, since the context of each appearance of an operand implies a type which can be checked against its declared type. Furthermore, this checking can be performed at compile time. However, proponents of dynamically typed languages argue that dynamically typed languages are more suitable for programming because of their flexibility and ease of implementation. However, such languages result in programs that execute slowly. More importantly, errors caused by operators having operands of the wrong type can only be detected at run time and then only if the code containing the error is executed, a condition that cannot be assured by testing.

Many "typeless" languages have been systems implementation languages, one of whose objectives is to utilize all the capabilities of the machine on which the language is implemented. Wirth [9] states that the most frequent reason for dealing with typeless data is the need to pack different kinds of information into a single word which the language regards as an indivisible entity. Wirth claims that such problems can be handled by including special-purpose constructs in statically typed languages such as PASCAL's packed records. Although these additions increase the volume of the language and complicate its compiler, they do not reduce the simplicity of the language. Not all designers of systems implementation languages agree on the need for typeless data. The designers

of the Systems Language for Project Sue claimed that type checking was the most useful single screen for logical errors within the project [2].

If a language is to have data types, the language designer must decide whether conversions will be explicit or implicit. Implicit conversions are probably more common in dynamically typed languages than in statically typed languages. However, while some dynamically typed languages provide implicit conversions when necessary and possible (e.g., 3 + '3' yields 6 in SNOBOL) not all such languages do (e.g., APL). Hoare [6] lists some of the disadvantages of implicit conversions:

(1) Incorrect results are often "nearly" right so that an error may be more difficult to locate
(2) The inefficiency of the conversion is often a shock
(3) The definition of the language is complicated by the rules for implicit conversions.

From the preceding arguments, it seems that statically typed languages with explicit conversions offer the advantages of abstraction mechanisms and the earliest and most reliable error detection. We would like to demonstrate that statically typed languages have this property.

III. EMPIRICAL EVIDENCE

We cannot logically prove that particular language features will enhance programming reliability, much less derive the amount of improvement by analysis. However, we can gather empirical evidence that tends to confirm or refute such claims by measuring the amount of improvement (or lack thereof) in real situations. We can observe programmers at work and examine the programs they create.

Some evidence can be gathered from an experiment which measured the effects of nine particular language design decisions [4]. The experiment involved observing the errors made by 25 reasonably experienced programmers (graduate and upper level undergraduate students, including part-time students with industrial experience) using two languages to write two rather small (60–150 lines), but fairly sophisticated, programs. The two languages were identical except for the nine altered features among which were statement punctuation; assignments; inheritance of environment; and expression evaluation. The programs, which involved concurrent operations, simulated a card game and a small spooling system.

Each subject was given a manual for the appropriate language and asked to learn the language himself. The experimenter was available to answer questions.

The listings of the subjects were examined by the experimenter for errors. On runs following the run on which an error appeared, a subject may change his program correcting the error or altering the manner in which the error manifests itself not a diagnostic message or incorrect problem output was produced the error runs, the error was said to persist if it remained uncorrected.

To determine the number of occurrences of the errors, they origin, and counted on all intervening runs. Errors that

traced back to the run in which the compiler first analyzed the source code. Errors that occurred in the input data were traced back to the run in which the program first reached execution of an input statement.

The persistence of an error is the measure of the number of runs in which an error occurred (i.e., from the origin of an error until it was either eliminated, by correction or removal from the program, or the solution to the program was submitted). Persistence is calculated by dividing the number of occurrences of errors by the number of errors. The existence of errors in final solutions to problems means that persistence is only a lower bound on the severity of these errors (i.e., they would have existed for at least one more run). This measure was also used to identify properties of PL/1 which are prone to human error [3].

IV. ERRORS INVOLVING DATA TYPES

In both languages, variables which can be assigned a value (scalars, array elements, and resources) are dynamically typed. The type of a scalar variable or array element is either integer or character depending on the type of the last value assigned to the variable. Of course, this implies that an array need not contain elements with homogeneous types.

A second type of operand in the languages is a resource, a combination of semaphores and queued values. In this paper, we shall only consider the queued-value aspects of these operands. The assignment operation is not defined for resource operands. Instead, the functions RELEASE and REQUEST manipulate queues of values.

RELEASE (Resource, Value)

places Value at the end of the queue named Resource.

REQUEST (Resource)

removes and delivers the first value of the queue named Resource. Resources have elements of both static and dynamic typing. By declaring an identifier to be a resource, a programmer restricts appearances of the identifier to the RELEASE and REQUEST functions and parameter lists. However, like arrays, the queue associated with a resource may contain both integer and character values. Despite the presence of statically typed features in the languages, no type checking is performed until execution time.

On 815 runs of programs, 3937 occurrences of 1248 errors were observed. Of these, 405 occurrences of 116 errors were attributable to operands having inappropriate data types. Table I contains a detailed summary of these errors.

Of the 116 errors, 104 (types 1–9, 12) occurring on 329 occasions involved operands whose data types could have been checked at compile time. Twelve errors (types 10–11) occurring on 76 occasions could only have been detected at run time. It is interesting to note the large difference in the persistence of these classes of errors: 3.16 runs for errors involving statically typed operands and 6.33 runs for errors involving dynamically typed operands. What kind of improvement could be expected from compile time type

TABLE I. Error summary.

ERROR	NUMBER	OCCURRENCES	PERSISTENCE
(1) Extra formal parameter list in procedure dcl	3	5	1.67
(2) Use of a statement as a function	9	9	1.00
(3) Omitted subscript list from an array reference	3	15	5.00
(4) Attempt to assign to a resource variable	3	10	3.33
(5) Attempt to fetch from a resource variable	8	46	5.75
(6) Incorrect type of resource parameter	10	23	2.30
(7) Case selectors of different types	1	4	4.00
(8) Incorrect number of parameters	47	144	3.06
(9) Conversion from character to integer	12	54	4.50
(10) Input data of wrong type	8	38	4.75
(11) Incorrect type of operand (dynamic causes)	4	38	9.50
(12) Other	8	19	2.38

checking? Of the errors involving statically typed operands, those in the first two categories (i.e., extra formal parameter lists and the uses of statements as functions) were syntactic errors with an average persistence of only 1.17 runs.

Among the errors involving statically typed operands was a CASE statement with both integer and character selectors. The CASE statement was implemented as a series of nested IF statements. This error was not detected until run time, when the selection expression was compared to both integer and character selectors.

```
CASE I OF
  1, 2: ...
  'A' : ...
  'R' : ...
END
```

Subjects also attempted to use the name of a resource to represent the first value in the queue, after they had used the REQUEST function to obtain that value. Apparently, the subjects had difficulty distinguishing between the queue and the instance of a value in the queue. For example, subjects would use the sequence of instructions

```
REQUEST (Resource);
OUTPUT (Resource);
```

rather than the correct statement

OUTPUT (REQUEST (Resource

The subjects also had difficulties with conversions. Conversion errors can be divided into two classes: those resulting from failing to convert a character value to an integer value; and those resulting from misunderstandings about the integer representation of character values. The only conversion available to the subjects was the XPL BYTE function which returns the integer representation of a single character argument (e.g., on the IBM S/370 BYTE ('9') returns 249). Thus, to convert a character to the equivalent integer, one must first apply the BYTE function to the character and then subtract the integer value of the character representation of zero (i.e., BYTE ('0')). Nine times, subjects failed to apply the BYTE function to character operands before using the results with operators requiring integer operands. These errors had a persistence of 3.89 runs. Perhaps the subjects believed that the "automatic conversion" offered by the assignment operator in a language with dynamic typing would be extended to other operators as well. In addition, 3 errors with a persistence of 5.67 runs resulted from subjects who failed to subtract the integer value of the character representation of '0' from their converted value.

Errors involving dynamically typed operands occurred when subjects inadvertently altered the type of existing operands. There are several methods of altering the type of an operand in the language: assignment operations; input statements; and parameter binding. Errors were observed which were attributable to each of these language features. In the following example, the RELEASE function places both character (e.g., Winner) and integer (e.g., Discard) values in the queue Message. The values from the queue are subsequently retrieved and compared to an operand (Winner) with a character value. Of course, the comparison will result in an error when an integer value is retrieved from the queue.

```
Winner:= 'GERONIMO';
REPEAT
   Discard:= 0;
   IF ···
      RELEASE (Message, Winner);
                  .
                  .
                  .
      RELEASE (Message, Discard);
                  .
                  .
                  .
ELSE IF REQUEST (Message) = Winner THEN ···
```

Another error occurred when a resource was bound to a formal parameter that a subject believed to be an integer. The parameter was used as an operand in a comparison, resulting in an error. A third error involved a subject who assigned the null character value to an array element and subsequently compared the values of each of the elements of the array to zero.

V. CONCLUSIONS AND CURRENT WORK

Although the evidence presented seems to indicate that statically typed languages offer better ab[illegible]ion mechanisms and error detection, it is by no means conclusive.

Therefore, we are preparing an experiment to compare the effects of a statically typed and a typeless language on programming reliability.

As was the case in the previous experiment, we shall design two programming languages, one with statically typed operands; and another with "typeless" operands. The two languages should be identical in all features not affected by the issue of data types (e.g., control structures, scope rules, and declaration requirements). In addition, the features of both languages should be shared with other widely-used languages so that the results of the experiment are easy to interpret.

The statically typed language should include at least two data types, preferably integers and strings, as well as arrays of data of each of the basic types. The "typeless" language should contain only words which the operators assume contain the data representation necessary to the operator. Thus, for example, an addition operator assumes that its operands are integers and a subscript operator assumes that its operand is an array.

It would hardly seem fair to provide the programmers working with the statically typed language with all the operations normally associated with string processing (e.g., concatenate, search, substring, length, blank trimming, and conversion) and to require that the programmers using the "typeless" language build these operations out of more primitive operations. If the statically typed language offered powerful operations in addition to abstraction and authentication, it would almost certainly aid programming reliability more than the "typeless" language. Instead, we shall restrict the string operations in the statically typed language to the essential string primitives: assignment; comparison; and selection of single characters within a string. This is an attempt to give the two languages "equally powerful" operations. Thus, programmers in each language will have to build the more powerful operations from sets of roughly comparable primitives.

The "typeless" language will have only single word variables. Programmers may associate an identifier with either a single word or a group of words. However, any identifier may be subscripted without error. In keeping with the idea of a "typeless" language, this decision allows the subscript operation to be applied to any identifier. A constant may have one of several representations, but its value must be storable in a single word. The numeric representations are binary, octal, decimal, and hexadecimal. Two character constant representations are provided: left-adjusted, blank-filled; and right adjusted, zero-filled. In order to manipulate the representations of data, a partial word operation may be applied to select or alter an arbitrary portion of a word (e.g., a character). In addition to the common infix arithmetic operators, there are three infix bit operators: AND; OR; and NOT. Stream input allows the constants of the language to appear as inputs and stream output produces decimal representations of values. Record input reads an input record (i.e., card) into twenty consecutive words (four characters per word) starting at the location specified by the actual parameter. Record output accepts pairs of arguments X, Y and prints the character representation of the Y words starting at location X.

The statically typed language will have integers, strings, and arrays of data of both types. However, the language will restrict strings and the operations defined on them in order to give the statically typed language and the "typeless" language equally powerful operations. Strings will be fixed length and will be padded with blanks on the right on an assignment or input operation in which the value being given to the string is shorter than the declared length of the string. Besides assignment, comparison, input/output, the only operation defined on strings will be the substring operation. The substring operation will be restricted to single characters within a string. T[illegible]

be no concatenation ... arch operation for strings in the language, although these operations may be built using substring and comparison. Integers will have the common arithmetic operations, but no bit operations. Only identifiers declared to be arrays may be subscripted. The constants of the language will be integers and strings. Stream and record input will be restricted to scalar variables (i.e., the programmer must write a loop if he wishes to read or write an entire array of strings). Stream input allows the constants of the language to appear as inputs and stream output produces the appropriate representation of a value. Record input and output deal only with scalar strings and values are padded on the right with blanks where appropriate.

Table II summarizes the different features of the two languages.

TABLE II. Typed vs typeless language.

FEATURE	STATICALLY TYPED	TYPELESS
Types	Integers Fixed length strings	Words
Constants	Integers Strings	Integers Left-adjusted, blank-filled characters Right-adjusted, zero-filled characters Binary, octal, and hexadecimal numbers
Structures Subscripting Substring	Arrays Arrays only Single characters of strings only	Groups of words Any word Any part of word
Built-in operations	No concatenate or search for strings No bit operations for integers	Bit operations: and; or; not
Input Stream Record	Any constant Single strings (80 characters)	Any constant 20 consecutive words (4 characters/word)
Output Stream Record Conversions	Any expression Strings None	Any expression Pairs of addresses and lengths None

Although it is obvious that both languages contain features common to many widely-used languages, the differences between existing languages prevent our simply picking two of them to use in this study. Conversely, implementing the two compilers from scratch would be an unattractive alternative. Therefore, we propose to alter an existing language and its compiler, SIMPL-T [1], to produce both compilers. SIMPL-T is procedure oriented, but does not have block structure. The language has only well-structured control constructs and its data types include integers and varying length strings, as well as arrays of both types.

There are several advantages to using the SIMPL-T language. SIMPL is the name of a family of languages which share common features. The family contains both a "typeless" language, SIMPL-X, and a statically typed language, SIMPL-T. The compilers for each of the members of the family have been extended from a compiler for their common base language and have been designed for easy modification.

Once the compilers have been altered, we shall proceed in a manner similar to that of the previous experiment. Manuals for each of the languages will be prepared. Student programmers will be asked to solve the same problem, first using one language and then the other.

There are several possible directions in which future work is possible. Additional experimental work with data types could involve comparing the statically typed language to another statically typed language with extended string operations (e.g., concatenation, multiple character substring, conversions, etc.) to discover the possible advantages of providing these high level operations. The statically typed language could also be compared to an identical language whose compiler/interpreter waited until run time to detect type mismatches. This experiment would point out the advantages of compile-time error checking.

If languages are to be designed to aid in the production of reliable software, much work needs to be done in both language design and the experimental evaluation of design decisions. Unless empirical evidence exists to justify decisions, language design will continue to be more of an art than an engineering discipline.

ACKNOWLEDGEMENTS

I would like to thank Professor J. J. Horning of the Computer Systems Research Group of the University of Toronto for his criticisms and suggestions concerning this work. The National Research Council of Canada and the General Research Board of the University of Maryland have supported this work. Computer time for the forthcoming experiment is being provided by the Computer Science Center of the University of Maryland.

REFERENCES

[1] V. R. Basili and A. J. Turner, "A Transportable, Extendable Compiler," *Software – Practice and Experience*, 5, 269–78 (1975).

[2] B. L. Clark and J. J. Horning, "Reflections on a Language Designed to Write an Operating System," *SIGPLAN Notices*, 8, No. 9, 52–56 (September 1973).

[3] G. Estrin, R. R. Muntz, and R. C. Uzgalis, "Modeling, Measurement, and Computer Power," *AFIPS Summer Joint Computer Conference Proceedings*, 34, 725–38 (1972).

[4] J. D. Gannon and J. J. Horning, "Language Design for Programming Reliability," *IEEE Trans. Software Eng.*, SE-1, No. 2, 179–91 (June 1975).

[5] J. V. Guttag, "The Specification and Application to Programming of Abstract Data Types," Computer Systems Research Group, University of Toronto, CSRG-59 (September 1975).

[6] C. A. R. Hoare, "Hints on Programming Language Design," Computer Science Department, Stanford University, STAN-CS-73-400 (October 1973).

[7] B. H. Liskov and S. N. Zilles, "Programming with Abstract Data Types," *Proceedings of a SIGPLAN Conference on Very High Level Languages*," *SIGPLAN Notices*, 9, No. 4, 50–59 (April 1974).

[8] J. H. Morris, "Types are not Sets," *ACM Symposium on the Principles of Programming Languages*, 120–24 (October 1973).

FAULT-TOLERANT SOFTWARE: MOTIVATION AND CAPABILITIES

H. Hecht

The Aerospace Corporation, El Segundo, CA

Fault-tolerant computers have been developed for applications that require a very high degree of hardware reliability, and it is frequently asked whether similar techniques can be brought to bear on software for critical applications. This question is addressed with particular emphasis on space-related software, both in space vehicles and on the ground. Some of the techniques described have much wider scope and will be of value to any critical real-time computer program. The principal techniques employed in hardware fault tolerance are seen to be applicable also through software fault tolerance: error detection; protective redundancy; and rollback provisions. Of course, they need to be implemented in a specific manner; particularly the redundancy must be provided by a different code than that used for the primary modules.

There exist today both an overall methodology for fault-tolerant software and specific techniques that can be utilized in implementing it. The recovery block concept (proposed by Randell) is particularly attractive for space-related applications because of its simple control structure. An acceptance test is performed on the primary module, and, if that should fail, a back-up module is called that furnishes the same output with a different program (and possibly from different input data). The recovery block, with the addition of a watchdog timer, has been implemented in a number of skeletor routines and has been found quite suitable in connection with the established structure for spaceborne software. A number of ad hoc fault-tolerance schemes can easily be incorporated in this control structure, e.g., use of a dead-reckoning module when the primary navigation module fails to receive data or produces otherwise unacceptable results.

A reliability model is proposed that shows a very considerable reduction in failure probability even when the fault-tolerance provisions themselves are far from perfect. Moreover, the software system failure probability is made less sensitive to the reliability of any given software module. It is, therefore, believed that the time is quite ripe to undertake serious studies of fault-tolerant software for space applications.

I. INTRODUCTION

Fault-tolerant computers have been developed for applications that require a very high degree of hardware reliability, and it is frequently asked whether similar techniques can be brought to bear on software for critical applications. This question is addressed here with particular emphasis on space-related software, e.g., ascent guidance software on launch vehicles, launch-control software for ground computers, and control and command software, both in space vehicles and on the ground. Some of the techniques described have much wider scope and will be of value to any critical real-time computer program.

In space applications software failures can cause loss of launch vehicles or [illegible] costing tens of millions of dollars. In the case of manned space missions the [illegible]ce

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976

possibility that a software failure might result directly in casualties. The need for highest reliability in both hardware and software segments of these computer systems is therefore obvious. In spite of the employment of the best available software development techniques and the participation of highly motivated and talented personnel, there have been serious software errors in space-related programs some of which led to spectacular failures, e.g., the launch failure of an early Mariner [1], and the destruction of a French meteorological satellite [2]. On the Apollo project, in spite of a tremendous expenditure on reliable software, computer programs caused some critical anomalies but, fortunately, a recovery procedure could be worked out in each case.

The principal techniques used in making computer hardware fault tolerant are concurrent error detection, protective redundancy (alternate modules accessible by switching), and rollback provisions that will return the program to an uncontaminated location after a failure. In principle, these techniques are applicable to software [3] but they need to be implemented in a specific manner. For hardware fault tolerance the backup modules are normally identical in design to the primary modules, but an identical copy of a computer program can hardly be expected to be of much help in recovering from a failure in the original. Therefore, redundancy in fault-tolerant software requires programs that are deliberately different from the original ones which they are intended to back up but that still perform the same function. Error detection and rollback provisions must be correctly adapted to be applicable to software failures. Examples of such techniques are described later, but it is not claimed that they are universally applicable.

The following assumes that the entire fault-tolerant software will reside on a single computer that has hardware fault tolerance transparent to the software execution. There are no major obstacles to other implementations, but by adhering to a specific baseline we can better explore the key issues in software fault tolerance.

II. MOTIVATION

Most organizations that develop software for essential space-related applications employ a combination of process controls and verification techniques. The process controls consist of programming standards manuals, periodic design reviews, and particularly some variant of the structured programming approach. The verification techniques include Independent Test and Evaluation (IT&E) and the use of automated test tools. The proof-of-correctness approach may also be considered, but it is not now in wide use for space applications.

The controls imposed on software development seem to be taking hold, and improvements in productivity as well as in conformance with schedules have been reported. It may be assumed that the reliability of the resulting software is also improved but specific proof of this, particularly oriented to space applications, is not yet available. While some satisfaction can thus be expressed regarding the development process there is continuing apprehension about the effectiveness of the verification techniques. The major shortcomings are:

(1) The existing verification methods do not indicate when test or review is complete.

(2) No existing method or methods can provide a guarantee of correct execution of a program.
(3) The best of the current methods are extremely costly.
(4) Bad data or environmental abnormalities can interfere with the execution of an otherwise perfect program.
(5) There is no generally accepted methodology for limited reverification after change of a program.

As a more speculative argument it may also be mentioned that nature seldom takes the single-string approach for the accomplishment of vital tasks.

The difficulty in establishing a clear termination criterion for test is by no means unique to the software field. "Program testing can be used to show the presence of bugs, but never to show their absence!" [4] could as well be said about many hardware items. But the lack of any creditable data on software reliability makes dependence on test particularly undesirable. Because these data are not available, the discovery rate of errors during test cannot be related to any observed reliability in actual use.

The insufficiency of existing methods to provide positive assurance of program reliability is explained in a recent study specifically devoted to that subject [5].

> "The principal methods for examining software products are formal testing and proof of correctness; of these techniques, formal testing is the most widely accepted. It is well known, however, that testing can only establish the presence of errors but cannot assure their absence. Testing also relies ultimately on manual skills to identify test cases and to evaluate results, and this can often have serious consequences on the reliable assessment of software."
>
> "Formal proof techniques have two main shortcomings. First, proof depends on the correctness of assertions against which the program is proved; inadequate or erroneous assertions can invalidate a proof however carefully it is carried out. Second, the proof of programs of any consequence often consumes more resources than are required to build the program itself; furthermore, the complexity of this process is such that the proof is frequently incorrect."

The most widely accepted approach to high reliability in space-related software is through Independent Test and Evaluation (IT&E) which adds 40–50% to the already very high development cost of these programs [6]. The actual coding accounts for only 20% of development cost, while testing by the developer usually accounts for about 50% of development cost. The total test costs (by the developer and the independent evaluator) therefore amount to five times the cost of coding the program. IT&E also has a serious schedule impact and requires a considerable effort on the part of the procuring agency in order to keep the developer and the IT&E contractor from getting in each other's way.

At the input and output the computer system interfaces with sensors, communication channels, and control mechanisms that are subject to failure, noise, and temporary abnormalities caused by the environment. Critical software is expected to cope with these conditions at least to the extent of preventing the propagation of local or temporary outages. In some cases (e.g., range safety programs) there is an explicit requirement to furnish output under abnormal operation of the primary data links. This requirement is being met today by various test and alternate data routines that are "patched" into the

single-string software structure, in effect making it locally fault tolerant. In adopting
all fault tolerance the requirement to cope with abnormal data input will be automatically implemented.

There is thus ample motivation to consider the development of a comprehensive methodology of fault-tolerant software for space-related applications. Before describing the capabilities that can be brought to bear on the problem, a few words may be said about the types of software programs that are addressed. At present critical space software executes on a uniprocessor, sometimes with a synchronous executive but increasingly now with an interrupt-driven executive. The critical software modules are those of the executive proper, applications programs servicing closed-loop control tasks (e.g., attitude control) and applications programs servicing open-loop tasks (e.g., staging sequences). In general those programs servicing closed-loop applications are more tolerant of temporary interruptions, or of a single erroneous output, than are the other routines. Software errors of short duration are similar to system noise and are attenuated by the feedback in these applications. The closed-loop routines constitute a large percentage of the critical
tware, and therefore their lower susceptibility to temporary program errors should be taken advantage of.

Parts of telemetry formatting, power scheduling, and similar housekeeping programs
usually not critical to spacecraft survival, at least in the short term. Because of the high cost of complete software fault tolerance it will be desirable to protect these less critical modules only to the extent that faults in their execution do not impact the essen-
application programs.

III. CAPABILITIES

There exist today both a.. overall methodology for fault-tolerant software and specific techniques that can be utilized in implementing it. The basic structures required for fault-tolerant software are a primary and an alternate module for each critical task, and a test for determining whether a module has executed correctly. The recovery block concept proposed by Randell [7] illustrated in Fig. 1 implements these basic requirements with a simple control structure, a very essential feature for fault-tolerant space-related software. Simplicity of control (the testing and alternate program routing associated with fault tolerance) and the applicability of the same control structure to all parts of an overall computer program assure that these elements can be tested so exhaustively that failures which would negate the desired program reliability can be virtually ruled out. This is in
trast to fault-tolerance provisions for network processor software where rather exten-
diagnostics and recovery routines have been proposed for each specific type of error or malfunction [8]. Randell also makes provisions for clear labeling of global variables as to "prior" and "current" values. Re-execution of a program after an error can make use of uncontaminated "prior" data.

Detailed techniques that can be brought to bear on fault-tolerant software have been described [9], and many are currently being applied in an ad hoc fashion to individual modules of essentially single-string programs. An example of use of alternate modules
be found in a navigation task where a switch to a dead-reckoning module may be

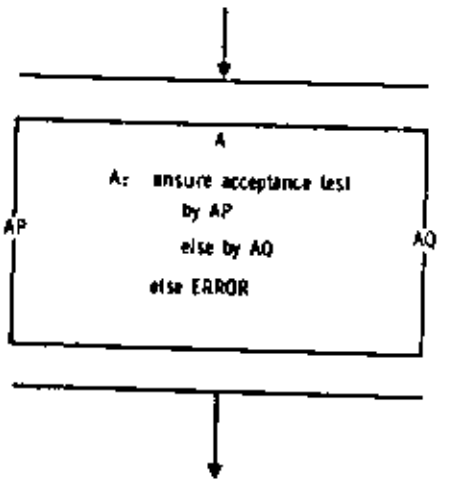


Fig. 1. Recovery block-Randell.

made when the primary module either fails to receive data or produces otherwise unacceptable results. A backup module for attitude control functions has been incorporated in the computer program of the Titan III space launch vehicle [10]. Backup modes for discrete events (shutdown, staging, etc.) are also implemented there. Although much of this work has been undertaken to cope with hardware faults, the resulting program has become partially software fault tolerant.

Input data for some space programs are checked for magnitude and data increments. Quantities such as velocity, temperature, or light intensity can vary over a wide range, but the values encountered at successive executions of a given program module are not expected to vary by more than a small amount, and a test based on this increment is therefore more sensitive. Parameters that cannot be depended on to vary in a smooth fashion are sometimes transmitted in redundant or in coded fashion to facilitate error detection on module entry or exit. Knowledgeable systems managers allocate adequate resources for this area because the most perfect software cannot be expected to give satisfactory results if it operates on incorrect data.

Another type of data screen uses the "reasonableness test" that checks consistency of related quantities at a given time or of the same variable at different times. An example of the first case is that if a gyro indicates a high spin rate for a vehicle, then there should also be a high rate of star transits. An example of the second case is that once the state of the vehicle has reached "liftoff" it would be unreasonable to expect it to be reset to "launch ready" at a later time.

Large commercial digital computers frequently have hardware traps for continuous monitoring of overflow or underflow, exceeding array bounds, and accessing unauthorized memory areas. The specialized hardware for these functions has in the past not been available in spacecraft computers because of weight and power limitations, but with the continuing trend toward miniaturization of computer logic elements this is expected to change.

IV. PROPOSED STRUCTURE

The following demonstrates the organization of these capabilities into a typical application routine of a fault-tolerant space software system. It is assumed that any recovery operations from hardware faults will be transparent to the program. The executive for this application consists of a normal segment that handles the routine communication with application modules, and an abort segment that is invoked when abnormal circumstances are encountered. Further detail on the executive is presented later.

In the implementation of Fig. 2, the normal executive formats two calls for a particular task, to the primary and backup application modules, and these preferably involve different calling parameters. Then the watchdog timer is set, and the primary call is issued.

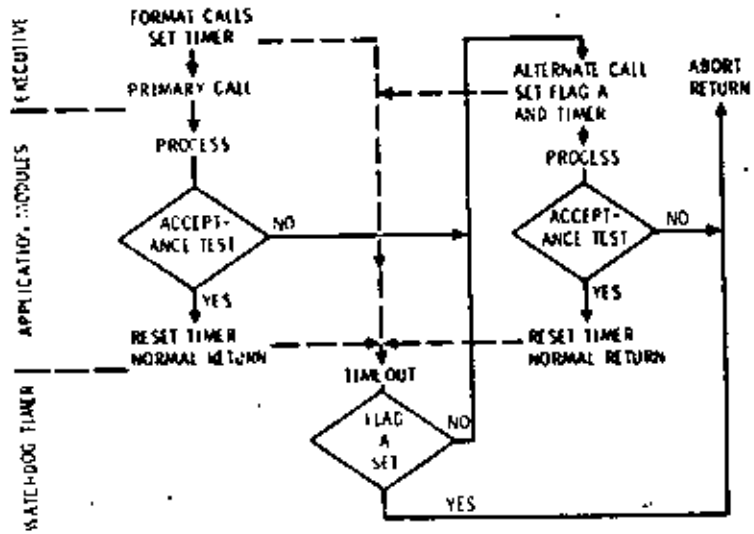


Fig. 2. Fault tolerance for application modules.

Upon success of the primary application module the timer is reset, and a normal return to the executive results. If execution of the primary application module fails for any reason a transfer is made to the alternate call, a flag is set, and the watchdog timer is reset. If the alternate application module executes correctly, the timer is reset and a normal return is made. The alternate application module may employ less accurate algorithms or utilize less precise data, but the return furnished by it must be of the same format as that returned by the primary module. The remainder of the application routines will then execute in a manner completely independent of difficulties encountered in the task identified here. If the return provided by the alternate module differs in format from the primary one, this would have to be recognized in later parts of the computer programs by additional branching. It is essential for a manageable control structure that one alternate program path not be dependent on execution of alternate program paths for any other application routines and hence the emphasis on a uniform return format.

For monitoring or telemetry purposes, the fact that the alternate program has been invoked is visible in the state of flag A. This flag is also essential to prevent continued looping in case a time-out is incurred during execution of the alternate module.

Failures in thoroughly checked out real-time systems are frequently caused by unusual timing or data dependencies. In that case, the next time the application routine in question is accessed the primary module may pass the acceptance test, and normal processing can resume. It is, therefore, not advantageous to make a permanent switch to the backup module after an isolated failure in the primary one. Where the time spent in repeatedly accessing a nonfunctioning primary module is objectionable a simple diagnostic, based on transitions of flag A, can be used to switch the roles of the primary and backup modules. In case of failure of the alternate module or in case of a repeated time-out, an abort return is initiated which will invoke the abort executive.

Figure 3 shows more detail of the primary application module. This expansion is intended to illustrate a number of optional implementations, and the control structure is therefore more elaborate than that of a typical application routine. It is seen that the

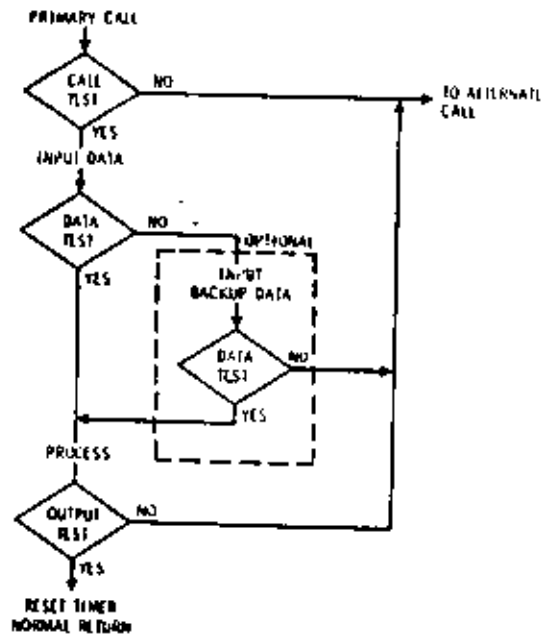


Fig. 3. Internal structure for primary application module.

single acceptance test in Fig. 2 may actually consist of a series of separate tests for correctness of the call procedure, input operations, and processing. Overflow, underflow, and other hardware-implemented tests may also be incorporated. Although the actual test structure is therefore more complex than that shown in Fig. 2, it has a single YES and a single NO exit and is a logical replacement for the simpler structure. The reason for the separation of the test blocks is that certain error-prone operations (primarily transfers of

data inputs) should be checked at the earliest possible time, both because the test can be more sensitive at that point and to prevent contamination of other program quantities. As has been mentioned in the previous section, these tests sometimes are incorporated in the current practices for very critical software programs. The significant feature of the fault-tolerant software concept is that the error exit from all tests goes to the alternate call, thus resulting in a simple and uniformly applicable control structure.

A variant of this general rule is shown in the optional backup data sequence in Figure 3. Use of this structure may be desirable where the primary data input is undependable or intermittent, where a source of backup data is readily available (e.g., primary input data from the previous cycle), and where processing via the alternate application module be disadvantageous. It should be observed that here again the primary data test and backup data test can be collapsed into a structure having a single YES and a single NO exit. The resulting structure corresponds to a recovery block within a recovery block, the need for which was foreseen by Randell [7]. If the backup data sequence is not to be used, the NO exit from the first data test will continue directly to the alternate call.

The purpose of the call test is to determine that the correct module has been reached and that calling parameters have been passed correctly. A suitable implementation is that the executive, in formatting the call, creates a check sum over the called address and the calling parameters. In the called module itself are stored (in a different memory location) its starting address and the locations where the calling parameters are expected. Checking over these locations, and comparison with the checksum received as part of the call, concludes the call test. Where the computer hardware provides extensive error-detection capability for memory access and readout, the call test may not be required.

The data test can address correctness of transmission, correctness of content, or both. To determine correctness of transmission any one of a number of error-detecting codes can be utilized, and some of these can be retained as an aid to fault diagnosis (primarily hardware oriented) in further processing. Checksums over blocks of data are, of course, also possible. To determine correctness of data content, the data type can be checked magnitude and increment tests described in the previous section can be performed.

The output test can employ the correctness features mentioned above, or explore correlation between input and output data. Where the application module serves as part of a closed-loop control system the special properties of such systems (e.g., expected smoothness of control) should be incorporated in the output test. Selective verification of program assertions may also be employed in this test [11]. If direct output to spacecraft equipment is to be furnished by an application module then this should be accomplished after successfully passing the output test and before returning to the executive.

A method for incorporating fault-tolerant application modules in the overall software system is shown in Figure 4. The block labeled "Application Modules" and the associated branching for normal and abort returns correspond exactly to the structure shown in Figure 2. In fault-free operation, the application modules are called by the task select section of the executive, and the normal returns are made again to that section. The same paths are still utilized if a primary application module fails and the alternate executes correctly. It should be remembered that normal returns from the primary and alternate modules must be of identical format.

If neither of the application modules executes correctly, the abort return path is entered and, as a first step, the failure is recorded. In manned spacecraft an appropriate warning message can be provided to the crew, and in unmanned spacecraft a transmission to ground control may be initiated. Then a diagnostic program is entered, which

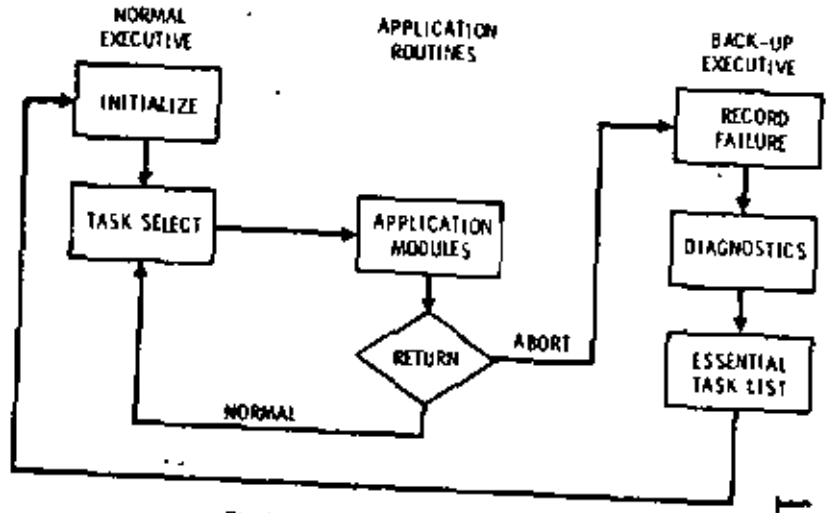


Fig. 4. Executive and application modules.

as a minimum should determine whether this is the first failure for the particular application module or whether it is a recurrence of a failure. Optionally, this program may also determine correlation of the failure with specified events (e.g., upload of the data package to this module). For some applications it may also be important to record the time interval taken up by the aborted application module. The diagnostic information of interest here is that necessary to formulate a recovery plan rather than to identify the specific cause of failure.

If the software failure was an isolated instance and if the time elapsed in executing the alternate module does not impact the accomplishment of other tasks, the diagnostic program may simply suspend the faulty application module and cause the task-select routine of the normal executive to step on to the next application module. If the failure has been diagnosed as a recurring one, or if the time elapsed in the abort procedure up to this point requires it, then a new task schedule has to be generated and substituted for the normal task list in the executive. This may take the form of an essential task list that is prepared a priori or it may involve a more selective procedure depending on the output of the diagnostic package in the backup executive. In either case the revised task list will be processed by the normal executive until either a self-contained decision criterion is reached or a crew or ground command is received that modifies this sequence.

In the discussion so far it has been assumed that an abort return is initiated when neither the primary nor the alternate module outputs of a recovery block can pass the acceptance test. However, the overall procedures can also be used to assure continued operation of the total software program after failure in a nonessential module for which no backup has been provided. As a minimum requirement, such a module should be monitored by the watchdog timer and hardware traps but it may also be followed by an acceptance test which provides an immediate abort return upon failure. From that point on the handling of an abort from such a module follows the procedure previously discussed. For isolated failures the normal executive will simply be caused to step to the next task, while for repeated failures the offending module can be removed from the task list.

The unified procedure for handling both essential and nonessential application modules is a very desirable attribute of this overall methodology.

V. RELIABILITY MODELS

The purpose of fault-tolerant software is to enhance the reliability over that obtainable from thoroughly checked out single-string modules. The following addresses modeling of fault-tolerant software such that the reliability improvement can be conceptually demonstrated and, given suitable input parameters, numerically assessed. Although the most basic procedures for reliability modeling are adapted from the hardware field, it is found that significant changes have to be made to make the resulting model applicable to software.

When a system consisting of a number of primary modules is provided with a backup or secondary module for each one of the primary functions, the resulting structure may be depicted as a series arrangement of paralleled modules. By applying conventional reliability calculations the failure probability of each module pair can be expressed as the product of the primary module failure probability and the secondary module failure probability. This does not recognize the difficulties in error detection and subsequent branching to the secondary module nor does it take into account the possibility of correlated failures in primary and secondary modules. The failure probability predicted by this model will therefore be much too low. Further, for a given overall software program the predicted reliability can be made almost arbitrarily high by making the modules small so that the individual module failure probabilities (which are in effect squared to arrive at system failure probability) become quite small.

The transition model shown in Fig. 5 has been devised to be representative of the failure processes that can be expected in typical space-related software applications. Starting at state 1, when the primary software module executes correctly, the transition possibilities over some arbitrary interval are the following:

(1) Primary software continues to execute correctly (path 1,1)
(2) Primary software fails and the failure is detected (path 1,2)
(3) Primary software fails and the failure is not detected (path 1,4).

In the first case, the module will definitely not cause system failure during the selected interval, and in the third case it definitely will cause system failure. For the case when failure is detected it has been found convenient to show transition to a pseudostate 2 from which an immediate further transition to either state 3 or 4 will result. The transition to state 3 represents the case where the backup module performs satisfactorily for at least one pass of the application program (i.e., the acceptance test is satisfactorily completed). The transition 2,4 represents the case where a failure has been detected but where the backup module does not satisfactorily complete the acceptance test. The probability that this will occur due to independent failures in the primary and backup modules is extremely low, but correlated failures, e.g., due to environmental effects or unforeseen hardware/software interactions, can by no means be ruled out. The transition 1,4 therefore models the event of correlated failure in the primary and backup modules.

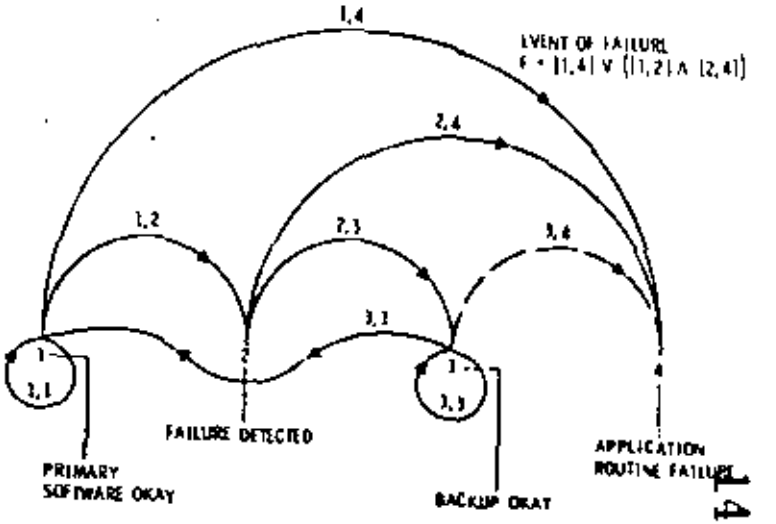


Fig. 5. Transition model for an application routine.

If the failure of the primary module is of a transient nature and if the backup module has performed satisfactorily, the executive will cause reversion to the primary module upon the next call to the application module, as modeled by transition 3,1. If the failure of the primary module is of a more permanent nature the diagnostics may reverse the roles of primary and secondary modules, and calls to this application routine will immediately bring the backup module into operation. Note that all of this is conditioned on the backup module having performed satisfactorily at least once after failure of the primary module. Any further failure of the backup module would therefore be an independent failure, and the probability of that event, while the primary module is still not yielding satisfactory performance, must be considered quite low. For this reason it is assumed that, once the transition 2,3 has taken place (backup module performed satisfactorily), it will remain in that state or revert back to the primary module (transition 3,1). The transition 3,4 cannot be completely ruled out, but the probability of this event will be several orders of magnitude less than that of the other transitions once state 3 has been reached. Therefore, the arc has been shown dashed and the probability of this transition is neglected in the discussion now following.

In this model the most significant events leading to failure of an application routine are seen to be inability to detect failure of the primary module (transition 1,4) and, given that primary failure has been detected, the correlated failure of the backup module (transition 2,4). Subject to previously stated conditions, the probability of failure of an application routine can therefore be expressed as

$$F_A = P(1,4) + P(1,2) \times P(2,4). \qquad (1)$$

A significant deviation from the hardware model has thus been established. Probability of failure for redundant hardware elements is modeled by considering failures independent, with probability of failure in detection rarely considered in elementary models.

For fault-tolerant software, on the other hand, these failure processes (dependency and inability to detect) are seen to be the governing ones. We have labeled these collectively "residual failure probabilities." They may be represented as an element in series with the independent failure probabilities of the associated application modules as shown in Figure 6. However, if our previous assumption is correct in that application routine failure due to independent failure modes of the primary and backup module is orders of magnitude less likely than that due to the residual failure modes, then a simplified model consisting entirely of a series arrangement of the residual failure modes may be constructed as shown in Figure 7.

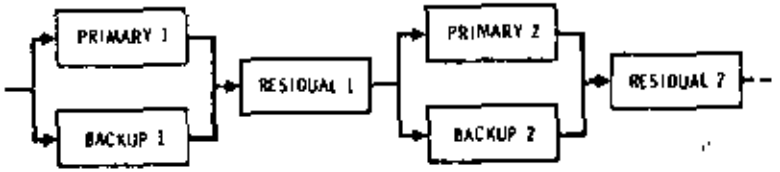


Fig. 6. Reliability Model.

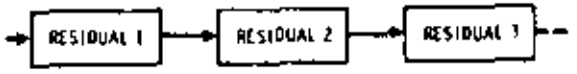


Fig. 7. Simplified reliability model.

The model developed thus far considers failure of any application routine (recovery block) as leading to system failure. This is implicit in the series arrangement of Figures 6 and 7. However, in Figs. 2 and 4 there is provision for an abort return when neither application module can pass the acceptance test. The primary purpose of this provision is to permit other application routines to continue to function after failure of one recovery block to permit resumption of a failed routine after a correlated transient failure or after ground intervention. This capability of the abort executive will certainly improve reliability in that a number of the cases which had previously been stipulated to lead to system failure may be completely recovered or may be converted to a degraded mode operation. Moreover, this capability of the abort executive to temporarily bypass application modules lends some degree of fault tolerance even to single modules. A simplified way of modeling this additional backup capability is shown in Figure 8. This model leaves something to be desired relative to the handling of failures not detected by the acceptance test. Since direct invocation of the abort executive does not take place then, it may seem inappropriate to indicate that any backup capability exists. In any case, however, the failure probability associated with the abort executive must be rated quite high, e.g., at 50% or more. Even at that level it represents a considerable improvement in system reliability. With this assumption and with the further understanding that some failures that do not initially cause violation of acceptance criteria may be detected later (e.g., by hardware traps), this model is still recommended for use in the quantitative discussion that follows.

The model developed here does not account for failures in the control structure used to implement the fault tolerance provisions nor in selected functions of the executive

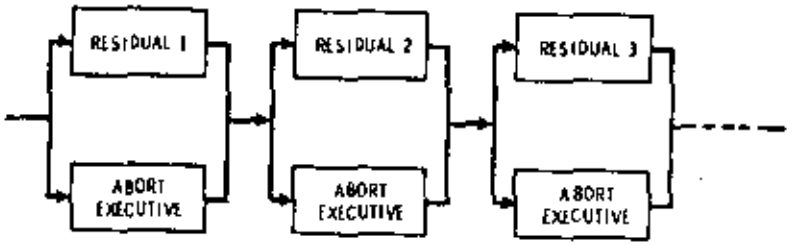


Fig. 8. Model of software system with abort executive.

(some functions of the executive, e.g., utility routines, can be protected by the same type of redundancy as the application programs). The control structure proposed here is so straightforward and capable of complete test (because of independence of control for one task on that for another) that its failure probability is considered negligible. The portions of the executive that deal with the normal and abort task scheduling may also be capable of very simple implementation, but this may have to be judged on a case-by-case basis. In addition, one may want to consider the possibility of a "wild" program overwriting essential portions of control or of the executive. This can be largely prevented by two means: permit write operations only after completion of the acceptance test; and provide only privileged write access to those portions of memory that contain the critical segments. Where it is felt that the failures due to singular events affecting control should be explicitly shown in the model, the structure of Fig. 8 can be preceded by a single block that represents these "higher order residuals."

VI. EXERCISING THE MODEL

There are at present few creditable estimates of software reliability and none at all for the effectiveness of the fault-tolerance provisions that have been discussed here. Nevertheless, some valuable insights can be gained by plugging *what-if* numbers into the model discussed in the preceding section. Fault-tolerance provisions are not likely to be incorporated in existing software that has performed adequately in the past and that is intended to be used essentially unchanged in the future. The baseline software reliability estimates that are of concern here are those for new or substantially modified software where a decision will be made whether to use a fault-tolerant approach or to engage in further testing. At that point, i.e., without the benefit of the extensive verification and validation procedures that are normally used for critical software, it is assumed that the a priori estimate for mission reliability of this software is between 0.95 (low reliability) and 0.98 (high reliability). Further, it is assumed that the total software program consists of ten application routines that will be structured into ten recovery blocks using the fault-tolerance concepts outlined above. The failure probability will be considered equally distributed among all the application routines so that the reliability for the individual routine will range from 0.998 (high) to 0.995 (low).

These assumptions form the basis for the calculation of application routine failure probabilities in Table I. A further set of assumptions for this table concerns the effectiveness of the fault tolerance provisions. From the preceding section it will be seen that

TABLE I. Application routine failure probability.

PRIMARY SOFTWARE RELIABILITY	HIGH			LOW	
FAULT-TOLERANCE EFFECTIVENESS	HIGH	LOW	HIGH		LOW
PRIMARY SUCCESS PROBABILITY, P (1, 1)	0.998			0.995	
FAILURE DETECTION RATIO, D	0.9	0.75	0.9		0.75
FAILURE DETECTION PROBABILITY, P (1, 2)	0.0018	0.0015	0.0045		0.00375
PROBABILITY OF UNDETECTED FAILURE, P (1, 4)	0.0002	0.0005	0.0005		0.00125
PROBABILITY OF CORRELATED FAILURE, P (2, 4)	0.2	0.4	0.2		0.4
APPLICATION ROUTINE FAILURE PROBABILITY, F_A	0.00056	0.0011	0.0014		0.00275
RELIABILITY INCREMENT, $\Delta R_A = 1 - P(1,1) - F_A$	0.00144	0.009	0.0036		0.00225

the principal deficiencies of the fault-tolerance provisions reside in inability to detect a failure or in correlated failure mechanisms of the primary and backup modules. In the absence of any experience it seems prudent to assign rather substantial probabilities to these failure modes. Accordingly, for highly effective fault-tolerance provisons, a failure detection success ratio of 0.9 and a correlated failure probability of 0.2 have been assigned. For fault-tolerance provisions of low effectiveness it has been assumed that the failure-detection success ratio is 0.75 and that the probability of correlated failures if 0.4.

The failure-detection probability and the probability of undetected failures are computed from these basic inputs as shown in Eqs. (2) and (3), where D is the failure-detection success ratio

$$P(1,2) = (1 - P(1,1)) \times D \ , \tag{2}$$

$$P(1,4) = (1 - P(1,1)) \times (1 - D) \ . \tag{3}$$

The probability of correlated failures is explicitly stated, and the total failure probability for an application routine (recovery block) with fault-tolerance provisions can then be computed by use of Equation (1). Of course, the lowest failure probability is obtained when the primary software reliability and the fault-tolerance effectiveness are both high. High primary software reliability and low fault-tolerance effectiveness lead to approximately the same results as low primary software reliability and high fault-tolerance effectiveness. Even where the primary software reliability is low and fault-tolerance provisions of low effectiveness are incorporated, the resulting application routine failure probability becomes respectable, being almost at the level of the high primary software reliability without fault tolerance provisions.

These conclusions are amplified by considering the total software system, including the operation of the abort executive. This is done in Table II where use has been made throughout of an approximate reliability calculation for the series system shown in Equation (4)

$$F_S = \sum_{i=1}^{n} F_{Ai} = 10F_A \ ; \ R_S = 1 - F_S \ . \tag{4}$$

TABLE II. Software failure probability.

PRIMARY SOFTWARE RELIABILITY	HIGH			LOW	
FAULT-TOLERANCE EFFECTIVENESS	HIGH	LOW	HIGH		LOW
SUCCESS PROBABILITY, PRIMARY APPLICATION MODULES ONLY, R_S	0.98			0.95	
FAILURE PROBABILITY, FAULT-TOLERANT APPLICATION ROUTINES, F_{SF}	0.0056	0.011	0.014		0.0275
SUCCESS PROBABILITY, FAULT-TOLERANT APPLICATION ROUTINES, R_{SF}	0.9944	0.989	0.986		0.9725
FAILURE PROBABILITY, FAULT-TOLERANT SYSTEM WITH 0.5 EFFECTIVE ABORT EXECUTIVE, F_{SFE}	0.0028	0.0055	0.007		0.01375
ΔR_S	0.0172	0.0145	0.013		0.01375
% ΔR_S	86	73	65		73

At the high module reliability involved here, the error introduced by the approximation is negligible. The effectiveness of the abort executive is its ability to temporarily bypass application routines for which the entire recovery block is faulty until either the conditions causing the fault have cleared themselves or until successful outside intervention has taken place. The assumption that this effectiveness is 0.5 is completely arbitrary but is believed to be on the conservative side. The failure probability of the fault-tolerant system with the 0.5 effective abort executive is computed by Eq. (5) where the factor 0.5 represents the failure probability of the abort executive

$$F_{SFE} = F_{SF} \times 0.5 \ . \tag{5}$$

The resulting entries in Table II show that a highly effective fault-tolerant system can produce an almost 90% increase in reliability (reduce the failure probability to 10% of its original value), while the less effective fault-tolerance provisions still produce an almost 75% increase in reliability. Further, when the action of abort executive is considered, the failure probability of a system with initially low software reliability to which low effectiveness fault-tolerance provisions have been added has an appreciably lower total failure

probability (0.0138) than the high reliability system without fault-tolerance provisions. Also the absolute difference in failure probability among all configurations considered here has been appreciably reduced. This indicates that fault-tolerance makes the system less sensitive to the state of the initial reliability as well as to the effectiveness of the fault-tolerance provisions.

VII. THE NEXT STEPS

We hope to have demonstrated here that capabilities exist for the implementation of fault tolerance in space-related software. Further, this capability is not due to any single breakthrough but is rather the result of combining and unifying elements of current practice in the field with the results of theoretical work that had been carried on in a more general context. The particularly attractive feature of the proposed implementation is the simple control structure which promotes error-free construction and full testability of the fault-tolerance provisions themselves.

We have not yet produced a detail analysis of computer resources (storage and execution time) that will be required for this approach. Recent and anticipated advances in hardware capabilities make restrictions in this area less significant. Moreover, it will be desirable to compare these resource requirements not only with an ideal single-string program that yields the same functions but rather with a practical program that incorporates at least some of the ad hoc fault-tolerance provisions currently in use in the field. This may be a time-consuming and more subjective task (because of the variations in current practice), but it is certainly capable of accomplishment and should be undertaken soon.

There remains, however, the judgmental question of how much saving in test and validation can indeed be achieved by fault-tolerant software. This question cannot be answered convincingly until two or three pilot projects have been completed. In at least one instance one might want to see a deliberate reduction in the software test effort so as to probe the true capabilities of fault-tolerant software. The effect of fault tolerance on program maintenance also needs to be evaluated. Considerable expenditures will be required in order to conduct realistic and convincing pilot programs, but these will be but a minor fraction of the expenditures now going into the development and test of critical single-string programs.

The reliability model shows that a very considerable reduction in failure probability can be achieved even when the fault-tolerance provisions themselves are far from perfect. Moreover, the software system failure probability is made less sensitive to the reliability of any given software module. It is therefore believed that the time is quite ripe to undertake serious studies of fault-tolerant software for demanding applications.

ACKNOWLEDGEMENT

The author wishes to thank Dr. E. D. Callender of The Aerospace Corporation and Prof. M. L. Shooman of the Polytechnic Institute of New York for their helpful review and comments.

REFERENCES

[1] "Equation Error Cited in Mariner I Failure," *Aviation Week*, 29 (August 6, 1962).

[2] "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's," CCIP-85, *1*, 3 (1972).

[3] A. Avizienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proceedings of the 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CH0940-7CSR, 459 (April 1975).

[4] E. W. Dijkstra, "Notes on Structured Programming," *Structured Programming* (New York: Academic Press, 1972).

[5] R. E. Kaiustead, "On the Feasibility of Software Certification," SRI Project 2385, Stanford Research Institute, Menlo Park, CA (1975).

[6] R. H. Thayer and E. S. Hunton, "Software Reliability—A Method that Works," *AFIPS Conference Proceedings 44*, 877 (May 1975).

[7] B. Randell, "System Structure for Software Fault-Tolerance," *Proceedings of the 1975 International Conference on Reliable Software*, IEEE Cat. No. 75CH0940-7CSR, 437 (April 1975).

[8] S. M. Ornstein et al., "Pluribus—A Reliable Multiprocessor," *AFIPS Conference Proceedings 44*, 551 (May 1975).

[9] S. S. Yau and R. C. Cheung, "Design of Self-Checking Software," *Proceedings of the 1975 Conference on Reliable Software*, IEEE Cat No. 75CH0940-7CSR, 450 (April 1975).

[10] R. V. Erlane, "Program 624A Flight Plan VII Guidance Equations," TOR-469 (5116-44)-14, Aerospace Corporation, El Segundo, CA (May 10, 1966).

[11] L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," *Proceedings of the 1975 Conference on Reliable Software*, IEEE Cat. No. 75CH0940-7CSR, 59-65 (April 1975).

SOFTWARE EFFECTIVENESS: A RELIABILITY GROWTH APPROACH

R. A. Pikul and R. T. Wojcik
Naval Underwater Systems Center, Newport, RI

The purpose of this paper is to provide justification that reliable software is a growth process through which many of the engineering techniques used on hardware programs can be effectively applied. Existing military standards, quality assurance plans, and system specifications are discussed as to their application to software. General program requirements which parallel MIL-STD-785A, "Reliability Program for Systems and Equipment Development and Production," are recommended. Data obtained from existing and developmental software projects are analyzed to provide a measure for reliable software.

Graphs, error rates, failure definitions and management overviews are also discussed as they apply to the data collection program. At present, statistical reliability techniques are not utilized since the various math models that have been reviewed were not considered applicable. The results of the study suggest four separate phases in the development of large software projects: (1) standardization and validation; (2) acceptance; (3) integration; and (4) operational demonstration. Acceptance criteria are estimated for each of the four phases of development. A discourse on formal test procedures is also presented.

I. INTRODUCTION

The effectiveness of military weapon systems presently being developed can be measured by their software and hardware availability, reliability, and performance characteristics. Hardware effectiveness can be measured by using existing reliability/maintainability engineering techniques, coupled with performance or capability test results, most of which have been developed as a result of the NASA space program. Many of the practices utilized in the development of reliable hardware can be directly applied to software development programs. However, fundamental differences in the inherent characteristics of software and hardware make it impractical to adopt all the concepts recommended for the development of reliable hardware. In order to achieve a measurement of system effectiveness, a reliability program for the software life cycle must be defined.

The purpose of this paper is to recommend an approach to the software life cycle that maximizes reliability growth. Throughout this paper, reliability growth is defined as the increasing probability of a system (hardware, software, operator) to perform required user functions under stated conditions. The three major goals of this paper are:

(1) To recommend a process that can result in reliability growth for software products

Presented at the Symposium on Computer Software Engineering
Polytechnic Institute of New York, April 20-22, 1976.

(2) To determine if the rate of growth can be extrapolated to predict operational reliability
(3) To determine the level of achieved software effectiveness and recommend criteria for accepting a software product in each phase of testing.

In conjunction with these goals, the paper defines certain software error terminology and recommends general program requirements that parallel those of MIL-STD-785A, "Reliability Program for Systems and Equipment Development and Production."

II. THE CONCEPT OF SOFTWARE RELIABILITY VERSUS HARDWARE RELIABILITY

During the past few years, there have been arguments as to whether or not hardware reliability techniques can be applied to computer software. Because of the inherent characteristics of software, many of the techniques used in evaluating hardware reliability are not applicable. Hardware is physical in nature and subject to failure patterns that are statistically measurable. Software is the transformation of a designer's ideas into a symbolic language for computer processing; therefore, its reliability is dependent on correct design and the expression of this design.

For the purpose of this paper, the terms software reliability and reliable software are interchangeable. Software is considered reliable if it satisfies the specification requirements. The functions it performs must be within the tolerances and conditions stated in the performance and the design specification. (The content of the performance and detail specifications is addressed after the next section.)

Some of the differences between software and hardware that affect reliability techniques are:

(1) Software modules, code, instructions, etc., do not degrade with time as a result of environmental stress or fatigue effects (i.e., wearout)
(2) Duplication of software products does not introduce new software errors
(3) A comprehensive failure mode and effect analysis is impractical for large software products because of the large number of distinct logic paths
(4) At the present time, detected errors do not provide information for predicting the remaining number of errors in a given program
(5) The correction of a software fault alters the configuration of the software and eliminates any possibility of its reoccurrence
(6) There is no standardized approach for exhaustively testing software in order to assure it meets all operational requirements.

The above list indicates the necessity for a new engineering approach in order to create reliable software.

A. Statistical Measurement

[illegible] introduces statistical techniques for evaluating software [illegible] the models were reviewed in the hope of finding an appropriate measure

that could be applied to our data. The search was further motivated by the fact that various program managers insisted that a Mean Time Between Failures (MTBF's) be used for software products. It appears that the MTBF concept is the only reliability measure management will accept. This is not a criticism of project management, but rather the realization that an effective reliability measure must be understood (accepted) by the average layman who is not expected to have a strong background in statistics. However, accuracy should not be sacrificed for simplicity.

Difficulty was encountered in applying most of the models reviewed. Many of the problems originated in agreeing on the assumptions made so that statistical distributions (e.g., poisson, exponential, normal) could be applied to interpret the software failure phenomenon. Some of the assumptions are:

(1) The errors remaining decrease monotonically and the error discovery rate is proportional to the number of remaining errors
(2) New errors are not introduced during debugging
(3) The software product failure rate (and associated hazard rate) is constant
(4) Software errors have the same likelihood of detection
(5) Software product data inputs are randomly selected
(6) Software errors are independent.

It was stated earlier that software does not degrade over time. Therefore, if all the errors were eliminated from a software product it would be perfectly reliable forever (i.e., probability of success is equal to unity). During the process of debugging, testing, and operation, it is desirable to achieve the perfect state for a software product, thereby providing a natural reliability growth. This reliability growth changes the distribution of times between failures. The reliability growth can be observed in the fluctuation of errors/operating time. Figure 1 is a histogram illustrating the percent of new errors over operating time. The trend indicated by the histogram is a decreasing error rate (reliability growth) as the operating time accumulates.

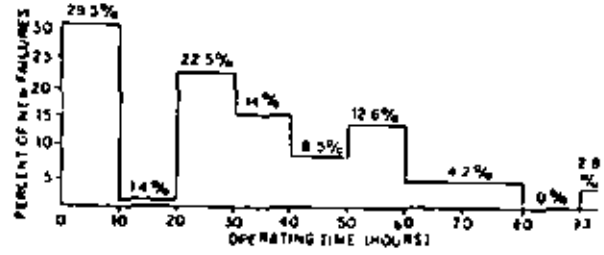


Fig. 1. Number of errors of cumulative operating time.

B. General Requirements for Reliability Programs

The military standard (MIL-STD-785A) includes the requirement for a reliability program, the specification of quantitative requirements, reliability demonstration, and a description of the program plan as part of the general requirements for [illegible] programs. In general, the quantitative requirements for hardware projects are [illegible] contractually as Mean Time Between Failures (MTBF) and Mean Time To Repair

(MTTR). The prototype equipment is subjected to reliability demonstration testing for the purpose of statistically estimating compliance with the quantitative requirements.

In addition, the Weapon Specification (WS-8506) outlines the documentation requirements for digital computer program specifications. This specification requires two additional documents: the Computer Program Performance Specification (CPPS), and the Computer Program Design Specification (CPDS). These are defined as follows:

(1) CPPS—The CPPS describes the performance requirements for the computer program portion of a given digital computer system. The CPPS contains performance criteria in terms of operational, functional, and mathematical language. The CPPS will be used by computer program design personnel and by personnel responsible for management, procurement, and maintenance of the computer program. Upon acceptance by the procuring agency, the CPPS becomes the base line document for configuration control of all subsequent programming efforts for the computer system.

(2) CPDS—The CPDS contains the design details for the computer program. It is prepared for the use of personnel responsible for the design, development, testing, and maintenance of the computer program and its subprograms and the personnel responsible for the procurement of the computer program. These personnel will have a general knowledge of the digital computer system architecture and programming. The CPDS is written in programming terminology and translates the functional descriptions of the performance specification into technical programming detail.

The reliability requirements for the software development program are contained in CPPS and CPDS.

Because of the conceptual differences between software and hardware, it is unrealistic to impose an MTBF requirement upon a software product. The idea of an MTBF requirement is all too often associated with the inverse of the failure rate because of the extensive use of the exponential failure distribution. The concept of Mean Time Between Failure is not suitable for software demonstration testing unless the underlying failure distribution is proven to be exponential.

Quantitative requirements for developmental software programs must be attainable and they must be measurable. A requirement on software that cannot be demonstrated is of little or no value (e.g., a minimum acceptable MTBF). The requirements can be expressed in terms of the frequency of errors (i.e., percentage of failures detected over accumulated operating time).

Unfortunately, there is no mathematical procedure that solves the problems of predicting minimum acceptable quantitative requirements for contractual purposes. As more attention is focused in this area and proper data collection methods are implemented, perhaps a mathematical procedure will be developed.

C. Schedule and Cost Considerations

It is necessary to consider scheduling and cost when specifying quantitative requirements. Figure 2 illustrates the software effort cost as it is related to reliability.

The total cost of the software effort is divided into the cost of designing, developing, and implementing the software product and the cost of maintaining the software in service. The in-service cost and design and development costs are inversely related (i.e., lengthy and frequently occurring down times due to software faults increase the in-service maintenance cost).

The total cost of the software effort is at a minimum when the sum of the maintenance in-service cost and total design and development costs are minimized. As shown in Fig. 2, the minimum total cost of the software does not correspond with high levels of

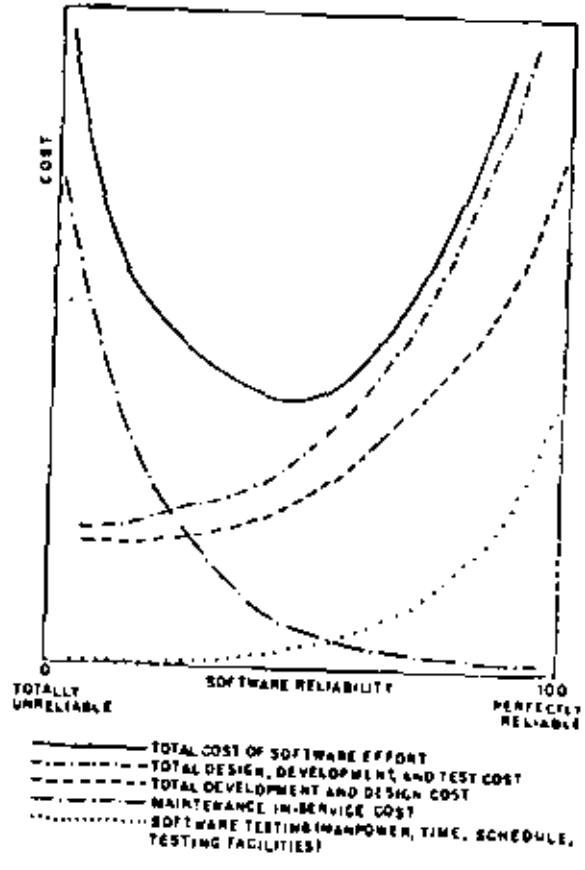


Fig. 2. Software development and in-service cost.

software reliability. For major programming efforts, the level of reliable software is directly related to the expenditures allocated for development, design, and testing. Emphasizing software reliability in the early phases minimizes the cost contribution of maintaining the software once it is released for use.

The selection of quantitative requirements specified in the CPPS and CPDS are dependent on cost and schedule. If the software must exhibit a high level of reliability, then allocations in the schedule and expenditures in the software effort are necessary. The sharp increase in the slope of the development and design cost and the software

151

testing cost denotes large expenditures for proportionally small increases in the level of reliable software. In establishing realistic quantitative requirements, the marginal increase in expenditures must be estimated

A similarity exists in the importance of the design between software and hardware products. If the design is poor, then the software product will achieve an acceptable level of reliability only when funds are allocated for debugging/testing or redesign [5, 6]. It is naturally cost effective to design a reliable product rather than relying on redesign to correct deficiencies.

The relationship of the software development schedule and the quantitative requirements is highly dependent on the individual development effort. The impact of schedule restrictions is evident in either a decrease in software testing and/or a decrease in the time allocated for development and design. Schedule constraints determine the maximum degree of software reliability that may be specified in terms of quantitative testing requirements.

Some of the factors that impact the scheduling for testing software are "facility utilization" and the hardware availability for testing purposes. The term "facility utilization" considers the environment available for software testing. In many instances, the software is tested by simulators in a laboratory environment rather than the actual hardware configuration.

Simulation testing also influences the schedule because of the additional time necessary to program the simulator. When the hardware is being developed simultaneously with the software, schedule delays in the developmental hardware are reflected in the software testing schedule.

The above discussion has touched upon a few of the cost and scheduling factors that must be considered in establishing realistic reliability requirements. It is of little value to demand high levels of reliable software that cannot be demonstrated because of schedule or prohibited cost.

III. DETAILED REQUIREMENTS FOR RELIABILITY PROGRAMS

The Military Standard (MIL-STD-785A) lists the following six major sections under the title of detailed requirements:

(1) Reliability management
(2) Reliability design and evaluation
(3) Reliability testing and demonstration
(4) Failure data
(5) Production reliability
(6) Status reports.

Some of these major hardware requirements are discussed below as to their applicability to software development.

A. Managing Developmental Software Programs

A major common characteristic of software and hardware is program management. The concept of software reliability impacts all levels of management, and it cannot exist without management support and participation. Reliability techniques for software and hardware can be implemented only with the backing of program management.

Large complex development programs can be managed efficiently by utilizing a systems concept to identify the proper function of each software segment. This concept is best defined as a systems approach where each manager comprehends his assignments (i.e., responsibility for individual tasks defined by work statements are integrated to the system level and not held at the autonomous level). The Chief Programmer Team [7] is a practical application of the concept to the programming function. The concept helps ensure that the problems arising in the development of software are identified early, thereby reducing the chance of not being detected until a crisis occurs. Each manager, regardless of his position in the hierarchy of management, views the program as a system and completes his individual task with the system in mind.

A common characteristic of large software programs is the simultaneous development of various subsystems. The classical approach to management organization stresses a scalar configuration. In such an organization, the managers at the same levels in the organizational structure communicate through the chain of command. The development of reliable software necessitates interfacing between managers and personnel having interrelated responsibility in the design effort. This lateral concept can facilitate modular programming techniques and reduce the number of problems during systems integration. Figure 3 is a simplification of a hypothetical software management structure. The lateral

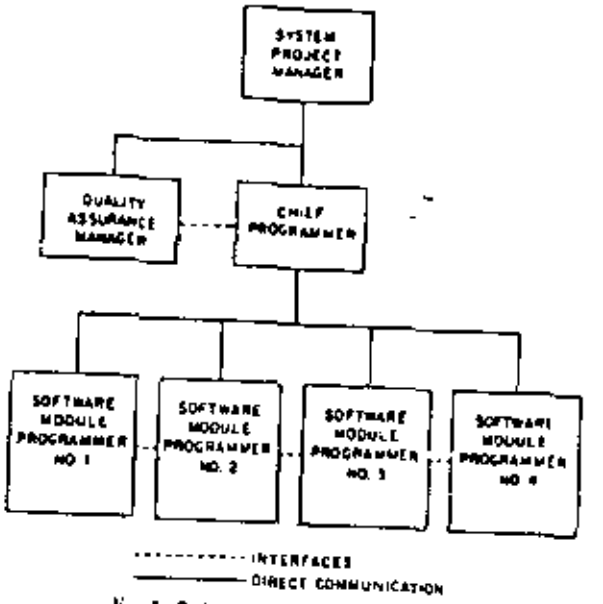


Fig. 1. Software management structure.

management philosophy is shown by the interfacing of the Software Module Programmers and the Quality Assurance Manager communicating with the Chief Programmer. The direct lines of responsibility provide the formal chain of command in such an organization.

Software development requires qualified personnel at all levels of management. In many instances the position is built around "key" personnel. The systems concept strives for all levels of management to be aware of program requirements so that a change in "key" personnel does not create unnecessary problems.

Management control is the proper application and review of requirements in the software development program and personnel management. The control function becomes simplified if the project planning phase is properly implemented. When detailed program requirements are not developed until the software is in coding or in late design, then management effectiveness is reduced. Without carefully established lines of communication, management control is weakened and the quality of the final software product becomes questionable. Management must realize that the software reliability concept is not an after-the-fact measurement tool, but rather an integrated aspect of successful software program management.

B. Reliability Design and Evaluation

The inherent differences between software and hardware renders many of the techniques applicable to hardware reliability programs useless in the development of reliable software. Stress derating, classical redundancy, and fatigue analysis do not apply to software. A significant difference in the quality evaluation of software and hardware projects is the application of statistical measurement techniques. However, there are some topics usually associated with hardware reliability program requirements that can be applied to software. These include the identification of mission profiles, critical time periods, high stress conditions, and failure definitions. These topics are fundamental to the establishment of realistic testing scenarios for proving program correctness.

At the present time, no proven mathematical technique has been established to predict the reliability of software products. Schneidewind [4] states that the form of the distribution between software anomolies may remain the same over time, but parameter values may change or the actual form of the distribution may change as test time accumulates. Software methods analogous to those of MIL-HDBK-217B, "Reliability Stress and Failure Rate Data for Electronic Equipment," simply do not exist because of the uniqueness of each software product. Historically, software development efforts do not include a comprehensive data collection program. The purpose of reliability prediction is to determine if the allocated quantitative design goals for hardware can be achieved. This technique is certainly desirable for software. However, the only method currently available to determine the achieved level of reliability for software is exhaustive demonstration testing.

C. Parts Reliability

Establishment of a preferred parts list is an integral part of the hardware development [illegible]. Programming languages and engineering techniques exist that can ensure

improved results for software. Requiring items such as computer languages, top-down programming, etc., may be considered analogous to requiring a quality level for components in a particular hardware product. Software development program management and designers determine which programming techniques result in the most effective software product. Depending on the management, programming staff, and performance requirements, the techniques recommended for software programs may vary significantly.

D. Failure Mode and Effect Analysis

The primary purpose of Failure Mode and Effect Analysis (FMEA) is to identify design weaknesses in the system and rank the deficiencies according to their effect on system operation. A comprehensive FMEA for large programming effort is not possible because all logic paths and corresponding inputs/outputs are not defined. Certain functions may be considered critical to mission effectiveness and safety. The software logic paths necessary to perform these functions can be flowcharted and reviewed in order to insure proper design.

E. Reliability Testing and Demonstration

Hardware reliability testing and demonstration is based on the application of statistical methods to estimate inherent equipment design characteristics. Because similar statistical techniques do not apply to software development programs, a new approach for demonstrating software is necessary. An approach that is consistent with the suggestions made up to this point stresses designing for reliability and testing the performance parameters defined in the specifications.

A primary goal of any software development project is to deliver reliable software which when called upon for use performs within defined limits. The limits as previously noted are a result of tradeoffs made by management and designers in the conceptual phase of program development to construct complete and unambiguous plans, milestones, and goals. The general philosophy in designing for reliable software is that the reliability of the software increases as the program is extensively tested (i.e., detected errors are corrected and then programs retested). It is assumed that the software is of the best possible design before the product enters the phases of formal testing. The major emphasis in software testing for reliability is the determination of acceptance criteria. In the following paragraphs, suggestions for software acceptance criteria are discussed.

A large scale software program containing large amounts of code and produced by many programmers can be considered to progress though four phases: validation; acceptance; integration; and operational demonstration. The goal is to define the optimal point at which the software product is ready to progress to the next phase. Therefore, the acceptance criteria are not developed solely for the operational demonstration phase, but they are determined for the three other phases as well.

Several analytical models have been suggested to fit software error rates [4] in the product life cycle. A study by Shick and Wolverton [8] relates the cost per routine and the number of instructions. Goodness-of-fit indicates that the relationship is very weak. The same study also indicates that the relationship between software problem reports and the number of instructions per routine is also poor.

Ramamoorthy, Cheung, and Kim [9] present an empirical model that indicates an increase in the error rate of a software program whenever it passes from one stage of development to the next. The shortcoming of this paper is the lack of suggestions for specifying realistic acceptance criteria for any test phase. Ramamoorthy and Ho [10] graphically relate the rate of detected software errors to the number of software program runs in development, see Figure 4. Also, Musa [2] attempted to fit a probabilistic

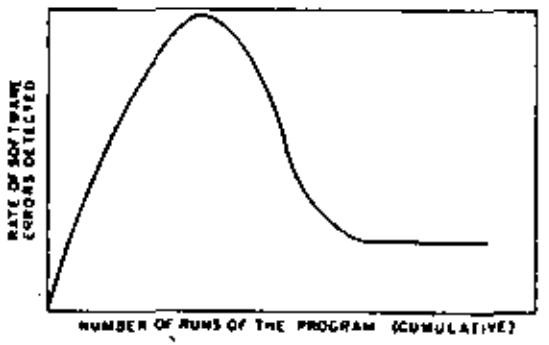


Fig. 4. Typical behavior of developmental software.

model for errors over program execution time. The model, through the graphical representation, exhibits a constant error rate being achieved after 25 hours of operation. Schneidewind [4] also concluded that trouble rates decrease and stabilize with testing. Once the error detection rate stabilizes, the software product can begin the next phase of testing or the current testing environment can be modified to increase the error detection rate (if possible). The environment is considered to be the input data selection process. To increase the stress for this type of environment, the selected input data attempts to exercise more of the logic paths and functional interactivity of the software. Errors are not detected unless the logic path containing the error is traversed.

Figure 5 is a plot of the errors reported on a commercial operating system at the Navy Lab over a 14 month period. The size of this software product is 2000k instructions. The plot emphasizes the weak relationship between software errors and calendar time; but it does not provide answers to the following questions:

(1) Is there any correlation between the number of users and the reported failures
(2) Do customers gain confidence in software as they utilize it more extensively and thereby increase the number of errors reported
(3) As the customer becomes more familiar with the software, does he recognize errors previously ignored?

Figures 1 and 6 are plots of failures during operating time for a developmental software program at the Naval Underwater Systems Center (NUSC). Figure 1 is a plot of operating time and new errors detected for the software package. Figure 6 is a plot of new failures (not previously detected or repetitive) and operating time that stresses or causes new activity for the software. Operating time is not included for the periods when the software is either [illegible] no new activity is initiated. As is expected, errors cannot be

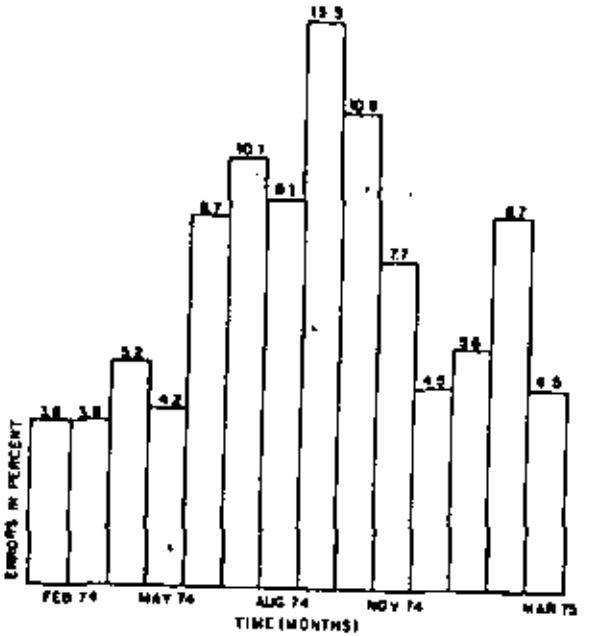


Fig. 5. Software errors vs calendar time.

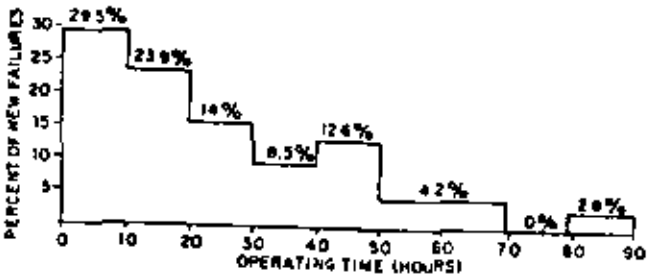


Fig. 6. Number of errors over censored operating time.

detected unless the logic path containing the error is executed. This is the basic factor that governs the design of software testing.

In the foregoing discussion, current data were presented (Figs. 1 and 6) in an attempt to state the assumptions for the development of acceptance criteria for demonstration testing. Figure 7 illustrates the approximate behavior of a software product in any phase. It indicates that the number of errors detected during any software development stage decreases with the number of runs (or total operating time). The number of errors never reaches a constant level because as errors are detected and corrected there is always the possibility of introducing new errors into the software. As the operating time increases, the error rate oscillates over some fixed value. The amplitude of the oscillation decreases [illegible] time and the only limit on this error rate is the software's operational obsolc[illegible]

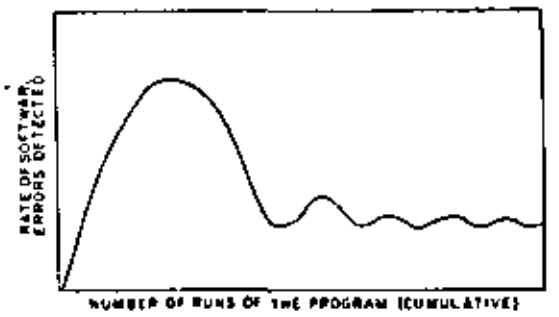


Fig. 7. Error rate approximation.

The expected number of errors increases over the initial operation of the system and finally stabilizes to a level (ϵ) after some period of time. As shown in Figs. 1 and 6, the time necessary to reach the apex of the curve is dependent on how exhaustive the testing is. Once the software progresses from one phase to the next (e.g., acceptance to integration), the number of new errors detected vs time plot is independently constructed. Therefore, a new plot is generated for each phase and an error rate estimate is required.

In order to estimate a realistic acceptance level, the operational environment must be clearly defined. The requirement, expressed as the ratio of errors to operating time, is established in the detailed specifications, and tests are devised to demonstrate the requirements. The significance of the test results is solely dependent on the extent of program exercise. The more logic paths and data input combinations exercised during the test, the more significant the test results become.

Unlike hardware, software does not degrade with time. Operating the program under the same condition and inputs does not uncover new errors. Strenuous mission scenarios provide the most useful form of software testing. Contrary to the ideas of Musa and Richards [2, 11], experience has shown that the inputs required for test scenarios are not selected randomly. Instead the inputs are chosen in order to measure the expected output. Any deviation from the anticipated output is a means of measuring product performance. Exhaustive testing would require randomly selecting all possibilities in the input range, which in large software projects may not be feasible because of time constraints on testing. Additionally, the possible number of input combinations may not be defined especially for real-time systems. The distribution of software errors over accumulated test time may be considered dependent on the input selection (test design). Also, testing done by an independent group is desirable so that the software demonstration testing measures software performance as defined in the performance and detailed specifications. When the testing is done by personnel who are directly involved in the software design and development, the danger exists that the test results will be biased.

If no knowledge is available on software tests of similar projects, then acceptance levels are developed by using engineering judgement. The software test does not only demonstrate an acceptance level for software, but is also a part of software design. Hardware demonstration tests are initiated after design, while software projects are ongoing during testing. This conclusion is drawn from the fact that once a software error is corrected, it will never manifest itself again.

F. Error Rates for Test Phases

Before recommending error rates for the four phases of testing, failure definitions must be considered. A clear distinction must be made between software design errors, whose symptoms appear infrequently, and intermittent hardware failures that induce software failures. Failures that cannot be attributed directly to hardware or software are difficult to troubleshoot. In fact, the failure may never repeat itself. Past experience has shown that the best way to eliminate these errors is to reload the system. However, for those failures that can be directly related to the software, the following definitions are provided:

(1) Software Fault – Is any discrepant operation of the computer program that does not require a computer reinitialization to resume normal computer operation.
(2) Software Failure – Is any discrepant operation that requires manual reload of all or part of the operational program with resultant loss of accumulated data. Software failures are further classified according to the extent of the reload.
(3) Recovery Time – For a software failure is that time required for a reload. The time necessary to re-establish the previous data base is not included.

The test plans and procedures necessary for demonstration testing should be prepared by the quality assurance group interfacing with the software development group. These procedures should reflect the requirements of the contract, the software specifications, and demonstration acceptance criteria. The plan for demonstrating achieved reliability should include the ground rules and criteria for deciding whether a test be classified as a success or failure, or whether the test be nullified due to invalid data, or due to other factors interfering with the established test conditions (e.g., hardware failures or invalid operational procedures). In addition, the test procedures should include the following details:

(1) A list and brief description of all modules and programs that will be placed on test.
(2) Test equipment to be used.
(3) Monitoring test equipment.
(4) Description of the hardware configuration necessary for software interface.
(5) Performance parameters to be measured.
(6) Performance parameters beyond which a failure has occurred.
(7) Failure classification for both hardware and software.
(8) Sample of report or log forms to be used.

1. Validation Test

The initial error detection process begins with the correction of obvious coding, keypunching, and language errors detected in the original compilation. Typical types of errors detected in the initial debugging of software are syntax errors, semantic errors, and the violation of high level language operating requirements. There are computer programs

available that detect coding configurations and misspecifications that cannot be detected by the high level language compiler. There is also hardware available that minimizes the human errors introduced in the keypunching process. However, most of the techniques described in current literature shed little knowledge on the detection of logic error in this early phase.

Based on observations from IBM Software and the CDC MASTER System, an estimate for the validation test is approximately seven weeks. In this time period an error rate (E) of 30 errors per operating week (168 hours) seems reasonable. Modules achieving this rate can begin acceptance testing. Modules that are produced late in the validation phase or still exhibit an unacceptable error rate continue in testing. The error rate for this phase focuses on all modules.

2. Acceptance Test

Acceptance testing is a functional checkout utilizing variable data to insure that the module or program is within specified limits. For example, a module constructed to add two numbers (2 + 2) that produces a result of 3,999 may not be within specified limits. Therefore, the module is rejected assuming that the hardware is not the limiting factor. The expected length of this test is estimated at three weeks. The expected error rate indicating that the software modules are ready to begin integration testing is five errors per 168 hours.

Some program managers prefer to designate this phase of testing as performance verification. However, it is generally agreed that the term "performance verification" relates to system level testing rather than component or module testing.

3. Integration Test

The third stage of testing developmental software is the integration test. The software modules are combined into the system and the system is mated with the designated hardware. The initial error rate in this phase is relatively high because the modules interact for the first time and errors that could not be detected in subsystem testing or by analytical methods arise. It is of interest to note that a system effectiveness measure can be estimated in this stage. This stage of software and hardware integration provides a measure of how often the system will be operational when called upon to operate for a specified length of time, and how "well" it did its job during that time (effectiveness measure). The expected length of integration testing, based on NUSC programming experience, is estimated to be six weeks. A realistic error rate is approximately 15 errors per week (168 hours).

4. Operational Demonstration Test

This test phase is either conducted in the actual environment for which the software was developed, or it is conducted by simulating that environment. At present, software product testing in this phase is conducted informally; that is, a producer releases the software with the guarantee of correcting any discrepancy noted. This effort requires the user to participate in a data collection and retrieval system. However, if a formal test is desired, it should include the actual hardware configuration for which the software was developed and performed with experience similar to that of the ultimate user, and

realistic scenarios that exercise the software properly. The expected time for completion of this test phase can be dependent on several factors (such as hardware availability, integration processing, numerical test requirements, etc.), thereby making the expected test time a variable.

Table I summarizes the four phases of testing discussed above. It should be noted

TABLE I. Summary of test phase scheduling and error rates.

Test Phase	Expected Time (weeks)	Acceptance Criteria (error rate) failures/week
Validation	7	30
Acceptance	3	5
Integration	6	15
Operational Demonstration	---	1

that these are engineering judgments based on IBM, CDC, and naval sonar development programs.

G. Configuration Control and Failure Reporting

Configuration control is the "systematic evaluation, coordination, approval or disapproval, and implementation of (1) initial configuration and (2) changes from that initial configuration." For hardware in the Navy, the vehicles used to control configuration are the Engineering Change Proposal (ECP) and ORDALT reporting systems. When a problem occurs and it is identified, corrected action is usually recommended through this system. The failure report contains the information necessary to describe the fault and to determine the fault location. Once the fault is corrected, all versions of the operational release package are updated. The advantage of this system is the controlling of updates to the initial configuration and providing life cycle support. A configuration control system for an operational naval software program is presently ongoing. Included in the data bank are the following records:

(1) Software master configuration history
(2) Active system problems
(3) Completed system problems
(4) Individual problem histories
(5) System modification status file
(6) Program change proposal index
(7) Program change proposal file.

IV. CONCLUSIONS AND RECOMMENDATIONS

The purpose of this paper was to identify areas that software and its reliability

programs have in common and to suggest ideas to ensure reliability growth. The major difference in the software development program is the absence of proven mathematical techniques for predicting the reliability of software. Because demonstration testing is an essential element in the software design process, the levels of acceptance must be defined in the CPPS and CPDS specifications. The requirements actually determine the extent of testing in each phase of software development. It is assumed that the testing exercises the maximum number of unique logic paths and input combinations to optimize the allocated time reserved for demonstration testing.

As previously stated, the intended function of software testing is to demonstrate a predefined level of performance. The inherent reliability that describes a software product is determined in the actual design and it is subsequently modified in debugging and testing. Software testing is necessary only because errors are introduced in design and development. Therefore, the major emphasis in the development of reliable software should occur in design. Software testing is intended to measure the reliability achieved in the design phase and if necessary to eliminate detected errors for modifying the existing level of reliability.

Once a software product is released for operational use, failure and problem reporting systems are necessary to ensure product life cycle support. The data chosen to be collected must be carefully weighed to determine its ultimate contribution to the software product. The cost involved in data collection, storage, and analysis restrict the type and amount of data available for statistical analysis. Very few software development efforts collect sufficient data for many of the proposed mathematical models. However, the data that is collected can be used as the basis for deriving future software quantitative reliability requirements.

REFERENCES

[1] P. B. Miranda, "Prediction of Software Reliability During Debugging," *1975 IEEE Annual Reliability and Maintainability Symposium*, New York, NY (January 1975).

[2] J. D. Musa, "A Theory of Software Reliability and Its Application," *IEEE Trans. Software Eng.*, SE-1, No. 3, 312–27 (September 1975).

[3] M. Shooman, "Software Reliability: Measurement and Models," *1975 IEEE Annual Reliability and Maintainability Symposium*, New York, NY (January 1975).

[4] N. F. Schneidewind, "A Methodology for Software Reliability Prediction and Quality Control," Naval Postgraduate School, Monterey, CA (November 1972).

[5] G. R. Craig, W. L. Hetrick, and M. Lipow, "Software Reliability Study," TRW Systems Group, Redondo Beach, CA (October 1974).

[6] L. J. Lapadula, "Engineering of Quality Software Systems," MITRE Corp., Bedford, MA (January 1975).

[7] F. T. Baker and H. D. Mills, "Chief Programmer Teams," *Datamation* (December 1973).

[8] G. L. Schick and R. W. Wolverton, "Achieving Reliability in Large Scale Software Systems," *Proceedings of 1974 Annual Reliability and Maintainability Symposium* (January 1974).

[9] C. V. Ramamoorthy, R. C. Cheung, and K. H. Kim, "Reliability and Integrity of Large Computer Programs," Electronics Research Laboratory, Memo No. ERL-M430 (March 12, 1974).

[10] C. V. Ramamoorthy and R. T. Ho, "Testing Large Software With Automated Software Evaluation Systems," *IEEE Trans. Software Eng.*, SE-1, No. 1, 46–58.

[11] L. R. Richards, "Computer Software: Testing, Reliability Models, and Quality Assurance," Naval Postgraduate School, Monterey, CA (July 1974).

Guest Editorial: Programming Environments

THIS Special Section was originally intended as a form of proceedings for a workshop on programming environments held in June 1980 at Schlumberger-Doll Research, Ridgefield, CT. At that workshop it became clear that the term "programming environment" meant many things to many people. The primary source of this confusion lies in different models of programming itself. To some, programming is concerned with developing large-scale software systems with long projected lifetimes; to others, programming is concerned with developing experimental software solely to help clarify new ideas. That is, different programming environments are aimed at different settings. Two aspects of a setting seem especially important when characterizing a programming environment:

1) programming in the large versus programming in the small
2) experimental software versus production software.

This is clearly illustrated by four programming environments which are currently in active use.

1) Interactive Lisp environments are used for developing experimental artificial intelligence systems and highly evolutionary operating systems. They generally include structure editors, prettyprinters, and interpreters with symbolic debuggers; some even include esoteric tools like automatic spelling correctors.

2) Fortran analysis tools are generally aimed at validating fairly large programs. These environments include such tools as static flow analyzers and facilities for inserting probes to test semantic correctness during execution.

3) UNIX™ is an operating system whose command language permits the combination of existing programs in a very flexible way. This has led to a style of interactive use which emphasizes program-using rather than program-writing. In addition, a variety of software tools have been developed, including the Programmer's Workbench which is aimed at supporting the development of large software projects.

4) PSL and PSA are tools for specifying certain kinds of large software systems and for analyzing such specifications.

Each of these environments was designed for a particular programming setting and tuned to facilitate software development within that setting. In trying to give a definition that encompasses all of these systems, perhaps all that can be said is that a "programming environment" is a computer-aided design system for software. It generally consists of a set of software tools, each aimed at assisting in some aspect of programming. Many environments are specifically designed for a particular programming language, methodology, or view of the programming process. In well-developed programming environments, this often leads to a certain kind of internal coherence and to a common style of use across large groups of users.

Perhaps the most important lesson we can learn about building programming environments is that it is inherently an experimental process. Our methodology must be one of building programming environments, testing them, and trying to abstract the lessons to be learned. In this process, we can both draw from and contribute to several of the traditional areas of computer science.

Programming Languages: Many parts of an environment are language-specific; interactive programming environments could significantly alter the way we think about programming languages.

Programming Methodology: Software tools could offer significant help in the use of particular programming methodologies; experiences with current programming environments could offer insight into alternative methodologies.

Software Engineering: Well-designed environments could have a major impact on software development; the entire software development process must be considered when designing an environment.

Artificial Intelligence: Some of the most innovative programming environments have been built in artificial intelligence centers; future programming environments are certain to have "intelligent tools" of some kind.

The papers in this section reflect this diversity. The paper, "An Incremental Programming Environment" by Medina-Mora and Feiler, describes a syntax-directed programming environment based on compilation technology. IPE is typical of the language-specific programming environments which are gaining popularity. The paper, "Why Programming Environments Need Dynamic Data Types" by Goodwin, describes some of the lessons about data types that can be learned from programming environments for languages with only weak typing. One important lesson is that the compilation and run-time environments should be less rigidly separated than is commonly done. The paper, "An Experiment in Small-Scale Applications of Software Engineering" by Boehm, describes the results of an experimental application of methodologies for programming in the large to a smaller scale project. The major lesson is that the utility of these methodologies is not restricted to large software systems. The paper, "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis" by Kant and Barstow, describes an experimental artificial intelligence system testing the feasibility of encoding knowledge about specific programming techniques in an automatic programming system. Their system suggests ways that future programming environments may be able to incorporate programming knowledge similar to that of a human programmer.

A small set of papers can neither summarize nor capture all of the important aspects of programming environments. Sev-

0098-5589/81/0900-0439$00.75

eral other collections of papers are (or will soon be) available; the interested reader is urged to consult them for more information about this area of emerging interest [1]–[4].

We would like to take this opportunity to express our gratitude to Erik Sandewall and Alan Perlis for working with us on the workship, to the National Science Foundation and the Office of Naval Research for financial support, and to Schlumberger-Doll Research for providing the facilities.

REFERENCES

[1] D. Barstow, H. Shrobe, and E. Sandewall, Eds., *Interactive Programming Environments*. New York: McGraw-Hill, 1981.
[2] H. Hunke, Ed., *Software Engineering Environments*. North-Holland, 1981.
[3] A. Wasserman, Ed., "Special issue on automated development support systems," *Computer*, Apr. 1981.
[4] —, *IEEE Tutorial: Software Development Environments*. IEEE Comput. Soc., 1981.

DAVID R. BARSTOW, *Guest Editor*
Schlumberger-Doll Research
Ridgefield, CT 06877

HOWARD E. SHROBE, *Guest Editor*
Massachusetts Institute
of Technology
Cambridge, MA 02139

David R. Barstow received the B.A. degree in mathematics from Carleton College, Northfield, MN, in 1969 and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA, in 1971 and 1977, respectively.

From 1977 to 1980 he was on the faculty of the Department of Computer Science, Yale University, New Haven, CT, where he is now an Associate Professor Adjunct. In June 1980, he joined the professional staff at Schlumberger-Doll Research, Ridgefield, CT. His research interests include artificial intelligence, expert systems, automatic programming, codification of programming knowledge, programming environments, and models of the design process.

Dr. Barstow is a member of the IEEE Computer Society and the Association for Computing Machinery.

Howard E. Shrobe received the B.S. degree in mathematics from Yale University, New Haven, CT, in 1968, and the M.S. and Ph.D. degrees for work on automated program understanding from the Massachusetts Institute of Technology, Cambridge, MA, in 1975 and 1978, respectively.

From 1968–1973, he worked in the computer industry as a Systems Engineer specializing in operating systems. He is presently a Research Scientist at M.I.T.'s Artificial Intelligence Laboratory. For the past two years he has conducted research on computer-aided design systems for VLSI.

Dr. Shrobe is a member of the IEEE Computer Society and the Association for Computing Machinery.

Persistent Software Errors

ROBERT L. GLASS

160

Abstract—Persistent software errors—those which are not discovered until late in development, such as when the software becomes operational—are by far the most expensive kind of error. Via analysis of software problem reports, it is discovered that the predominant number of persistent errors in large-scale software efforts are errors of omitted logic. . . , that is, the code is not as complex as required by the problem to be solved. Peer design and code review, desk checking, and ultra-rigorous testing may be the most helpful of the currently available technologies in attacking this problem. New and better methodologies are needed.

Index Terms—Complexity, omitted logic, persistent software error, research in the large, software problem report, testing rigor.

Introduction

IT IS well known that software errors vary in expense. That is, software errors which are found quickly and easily, such as syntactic errors and blatantly catastrophic errors, are detected and corrected at little cost (see Fig. 1). On the other hand, those errors which elude normal software review and debug practices, and persist into the software operation/maintenance phase, may be quite expensive.

The expense connected with such errors lies partly in the cost to detect, partly in the cost to correct, and partly in the cost of an inoperable or unsafe software product. Although the first two costs are important, the third is far and away the most significant. Especially in embedded computer systems, such as those controlling aircraft in flight, or a rapid transit vehicle, or a spacecraft, software error cost may be measurable in lives as well as dollars.

Little has appeared in the literature distinguishing between errors by cost. Tools and methodologies for the detection and correction of software errors are proposed and advocated independent of their value in identifying high-expense versus low-expense errors. Software reliability practices and software reliability research which focus on this dichotomy would appear to have large payoff. This paper reports on a study which is an initial effort in that direction.

This study seeks to better understand "persistent" software errors. An error is defined to be persistent if it eludes early detection efforts and does not surface until the software is operational.

Manuscript received July 31, 1979; revised September 2, 1980.
The author is with Boeing Aerospace Company, Seattle, WA 98124.

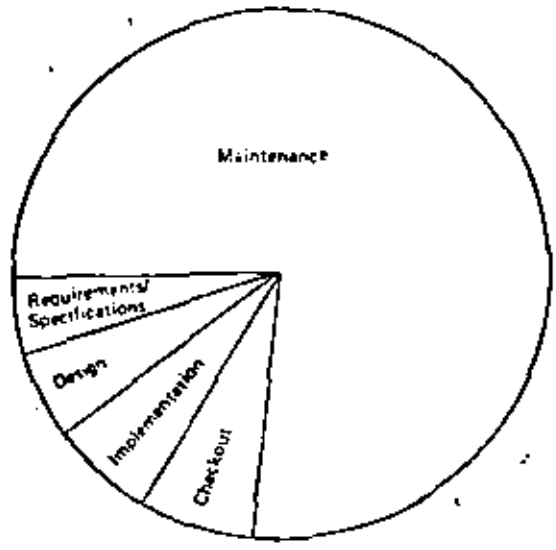


Fig. 1. Software life cycle: per error fix cost per phase.

The Study

In order to study those kinds of errors, two significant and mature software products were analyzed. Both are operational software systems for military aircraft use. Project A involved 150 programmers at the peak person-load, and contains about a half million instructions in the operational software alone. Project B involved 30 programmers and about 100 000 instructions. Thus, these software products may be considered to be typical of the state-of-the-art in large embedded computer system software.

The size of these software products is important. The point has frequently been made in the literature that large software systems and small software systems are entirely different, and that research "in the small" (using small programs or data/people populations) cannot be extrapolated to be equivalent to research "in the large" [1]–[3], [5], [6]. This study is an example of research in the large; no other approach is likely to be meaningful in the world of large, significant software products.

The method of approach in this study was to examine project-specific software error reports. State-of-the-art methodology in embedded computer systems calls for the filing of a software problem report (SPR) for each software error detected. The report provides spaces for three categories of information: 1) a symptomatic description of the problem from a user point of view, 2) a description of the problem from an internal software point of view, and 3) a description of the software correction. See Figs. 2–4.

0098-5589/81/0300-0162$00.75 © 1981 IEEE

SOFTWARE PROBLEM REPORT

SPR No. _____

PROBLEM: (Prepared by User)

Originator's Name _____ Organization _____ Phone No. _____

System, Processor, or Component Failing or Project Involved _____ Computer _____ System Version ID _____ Test case or Program ID _____

Description of Problem: The check for a valid platform in task PMSD is illogical. The software checks the data base for non-zero to determine valid platform. The software must expand its logic to determine if the platform-destination combination is valid.

Classification
- [x] Error
- [ ] Information
- [ ] Revision Request

Correction Required by Date _____ Reference UER No. _____

Authorizing Signature _____ Organization _____ Date _____

ANALYSIS: (Prepared by organization responsible for software)

Received Date _____ Time _____

- [ ] Coding Error
- [x] Design Error
- [ ] Software Not in Error, Explain
- [ ] Error Previously Reported On SPR No. _____
- [ ] Others, Explain

Explanation: Insufficient brain power applied during design.

Documentation Impact Milestone
1 [ ] 2 [ ] 3 [ ] 4 [ ]
5 [ ] 6 [ ] 7 [ ] 8 [ ]

Signature _____ Organization _____ Date _____

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: Modify code to check if destination is serviced by platform received on input data, or by both platforms.

Run Module Test PM3.1 (completed)

Mod/Programs Changed _____ Hard Load [ ] Yes [ ] No

Work Performed by (Signature) _____ Date _____

CONFIRMATION: Corrections Verified by Product Assurance

Signature _____ Date _____

MTM No.(s) _____

Available in (Version ID) _____

WHITE = Originator Copy GREEN = Analysis CANARY = Originator Closed PINK = Product Control Closed GOLD = Product Control Open

Fig. 2. Omitted logic.

Typically, large software efforts spawn hundreds or even thousands of such reports. SPR's are filed because of real software errors; because of problems which turn out to be errors not caused by software (e.g., computer hardware errors); and for changes which are desired by the user but are not errors. Only the first category of problems—real software errors—was examined in this study.

The SPR's were studied in "raw" (handwritten report) form. Every attempt was made to utilize the information as the programmer reported it, in order to eliminate deletions or transcription error which result from clerical encoding of the information, such as for a computerized database.

The thrust of the study was to divide these SPR's into categories, in order to identify the type of errors which are most

162

SOFTWARE PROBLEM REPORT

SPR No. _____

PROBLEM (Prepared by User)

Originator's Name _____ Organization _____ Phone No. _____

System, Program, or Component Failing or Project Involved _____ Computer _____ System Version ID _____ Test case or Program ID _____

Description of Problem: Maintenance task MSSW must be queued to operate at the last regular PD on the arrival ramp instead of the first ramp PD.

☐ Clarification
☒ Error
☐ Information
☐ Revision Request

Correction Required by Date _____ Reference UER No. _____

Authorizing Signature _____ Organization _____ Date _____

ANALYSIS: (Prepared by organization responsible for software)

Received Date _____ Time _____

☒ Coding Error
☐ Design Error
☐ Software Not In Error, Explain
☐ Error Previously Reported On SPR No. _____
☐ Others, Explain

Explanation: MSSW needs to execute at the last PD where tracking is performed by PPAM to avoid false anomaly reporting by PPAM due to a change in the vehicles assigned point.

Documentation Impact Matrix

1	2	3	4
☐	☐	☐	☐
5	6	7	8
☐	☐	☐	☐

Signature _____ Organization _____ Date _____

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: Revise task table data in PSECT SFPDTK to cause MSSW to be queued at PD M32 instead of M30.

Mod/Programs Changed: MESTA

Hand Load ☐ Yes ☐ No

Work Performed by (Signature) _____ Date _____

CONFIRMATION: Corrections Verified by Product Assurance

Signature _____ Date _____

MTM No.(s) _____

Available in (Version ID) _____

WHITE = Originator Copy GREEN = Analyst CANARY = Originator Closed PINK = Product Control Closed GOLD = Product Control Open

Fig. 3. Referenced wrong data variable.

prevalent. Here, an unusual approach was taken. Although error categories are well-known in the literature—TRW developed a pioneering software error category system [10]—those categories were not used in this study. Instead, the errors were allowed to "self-categorize." That is, as each SPR was reviewed, either it was assigned to 1) a category which described its own nature or 2) a category self-generated by some previous error.

There is some controversy attached to the use of raw SPR's and the use of self-categorization.

Regarding raw SPR's, the best approach—that is, the one closest to complete knowledge of the error—would appear to be actual review of the erroneous code and the correction. This has been used by Howden in his error data studies [7]. Use of raw SPR's, however, is an accurate and reproducible approach, since the SPR form repositories are typically sub-

SOFTWARE PROBLEM REPORT

SPR No. _____

PROBLEM: (Prepared by User)

Originator's Name _____ Organization _____ Phone No _____

System, Processor, or Component Failing or Project Involved _____ Computer _____ System Version ID _____ Test Case or Program ID _____

Description of Problem: A non-numeric character on the schedule tapes is correctly rejected. However, the disk schedule index file is not completely updated in this case.

Classification
- [x] Error
- [ ] Information
- [ ] Revision Request

Correction Required by Date _____ Reference UER No. _____

Authorizing Signature _____ Organization _____ Date _____

ANALYSIS: (Prepared by organization responsible for software)

Received Date _____ Time _____

- [ ] Coding Error
- [ ] Design Error
- [ ] Software Not in Error, Explain
- [ ] Error Previously Reported On SPR No. _____
- [ ] Others Explain

Explanation: Incorrect output ESR upon detecting non-numeric key.

Documentation Impact Milestone
- [x] 1 [ ] 2 [ ] 3 [ ] 4
- [ ] 5 [ ] 6 [ ] 7 [ ] 8

Signature _____ Organization _____ Date _____

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: Adjust output character count for digit error schedule index output ESR to output the entire file.

Mod/Programs Changed: C1ISF _____ _____ _____ Hand Load [ ] Yes [ ] No

Work Performed by (Signature) _____ Date _____

CONFIRMATION: Corrections Verified by Product Assurance

Signature _____ Date _____

MTM No.(s) _____

Available in (Version ID) _____

WHITE - Originator Copy GREEN - Analyst CANARY - Originator Closed PINK - Product Control Closed GOLD - Product Control Open

Fig. 4. Omitted logic.

o configuration management practices. The decision to w SPR's in this case was purely pragmatic—a large num- f errors can be reviewed rapidly with minimal loss of nticity

arding self-categorization, the study was built on the se of exploring new ground. That is, to the author's ledge, no one has studied persistent software errors per se e. To avoid the possibility that traditional error categories were not appropriate to persistent errors, the traditional categories were deliberately avoided. Self-categorization, of course, has the flaw that the judgment of the individual researcher will have some impact on the final results. However, the traditional categories discussed in [10] are ambiguous enough that they share this problem.

In any case, 100 software errors from each of two projects were subjected to review at the raw SPR level via self-

TABLE I
PERSISTENT SOFTWARE ERRORS BY FREQUENCY OCCURRENCE (200 ERRORS EXAMINED, 100 ON EACH PROJECT)

Category:	Project A	Project B	Total:
1. Omitted logic (existing code too simple)	36	24	60
2. Failure to reset data	17	6	23
3. Regression error	5	12	17
4. Documentation in error (software correct)	10	6	16
5. Requirements inadequate	10	1	11
6. Patch in error	0	11	11
7. Commentary in error	0	11	11
8. IF statement too simple	9	2	11
9. Referenced wrong data variable	6	4	10
10. Data alignment error (leftmost vs. rightmost bits, etc.)	4	3	7
11. Timing error causes data loss	3	3	6
12. Failure to initialize data	4	1	5
13. Other categories of lesser importance (total 4 or less) - Logic too complex, compiler error, data storage overflow, expression incorrectly coded, pointer one off, dynamic allocation failure, data not included in checkpoint, microcode error, data boundary problem, macro error, multitasking synchronizing error, erroneous initialization, naming conventions violated, logic order incorrect, interface mismatch, data reset in error, parameter mismatch, inefficient code, data declaration wrong, bad overlay, statement label at wrong place, data clobbered.			

NOTE:
An error was allowed to tally in more than one category. "failure to reset data", for example, is almost always a specific instance of "omitted logic." So are "if statement too simple" and "failure to initialize data." Any error could also be a "regression error."

categorization techniques. The errors were considered to be persistent on the basis that they were the most recent errors detected on those production-status projects. In most cases, this proved to be a sufficiently valid basis. In some cases, however, the errors were spawned by the correction of other persistent errors (these are commonly called "regression errors"). Errors of this class were included in the study on the grounds that such regression errors are just as costly as nonregression persistent errors; however, these errors were allowed to self-categorize a category for themselves.

An error was allowed to tally in more than one category. No attempt was made to provide mutually exclusive categories; the emphasis was on creating a realistic summary of the data as it was analyzed, and not to force it to fit an externally applied artifice. For example, the SPR in Fig. 2 would have been categorized both as "omitted logic" and "if statement too simple."

The process of analysis, then, was simply this: 1) project-specific, configuration-managed SPR forms, filed in chronological sequence, were examined one at a time; 2) using the "problem," "analysis," and "correction" information (see Figs. 2–4) the nature of the error was ascertained; 3) that error was categorized into its own and/or a previously selected category; 4) a tally was added to those categories. At the conclusion of analysis of a set of project-specific SPR's, tallies for the (variable number of) categories were summed. The categories for one project overlapped partially but not entirely with those of the other project (e.g., in Table I note that project A had no patching or commentary errors. Presumably project A did not allow patches and did not file SPR's on commentary errors).

THE FINDINGS

The findings of this study appear to be significant. That is, the categorized persistent SPR's show a consistent and definite pattern (see Table I).

The major finding of the study is that a large percentage of persistent software errors are instances of the software not being sufficiently complex to match the problem being solved. It is as if the programmer mind is straining to handle the complex interrelationships of a problem solution, and has failed. For example, a large number of such errors are the result of a predicate not having enough conditions—some flag or piece of data was not taken into account when it should have been—or of a variable not being reset to some baseline value after a major functional logic segment has finished dealing with it.

Here again, it is important to distinguish between the large problem and the small problem environment. Intuitively, it is easily seen that this kind of error is much more likely to emerge in the large rather than the small problem. The interrelationships between data and logic are much more entwined and complex in the large problem environment. It has even been said that "a 25 percent increase in problem complexity leads to a 100 percent increase in program complexity" [11]. And, in fact, most professional programmers have built software which they then realized was, in some areas, beyond their ability to comprehend (in the sense that its results were not predictable prior to its execution).

These problems of complexity may be considered to be design errors, and indeed many of them are. It is well known that design errors dominate the population of software errors (e.g., [4]). However, it must be recognized that they are the kind of design error which occurs at the most detailed level,

SOFTWARE PROBLEM REPORT

SPR No. /

PROBLEM: (Prepared by User)

Originator's Name _____ Organization _____ Phone No _____

System, Processor, or Component Failing or Project Involved _____ Computer _____ System Version ID _____ Test case or Program ID _____

Description of Problem: A non-numeric character on the schedule tapes is correctly rejected. However the disk 'schedule index file is not completely updated in this case

Classification:
☒ Error
☐ Information
☐ Revision Request

Correction Required by Date _____ Reference UER No. _____

Authorizing Signature _____ Organization _____ Date _____

ANALYSIS: (Prepared by organization responsible for software)

Received Date _____ Time _____

☐ Coding Error
☐ Design Error
☐ Software Not in Error, Explain
☐ Error Previously Reported On SPR No. _____
☐ Others Explain

Explanation: Incorrect output ESR upon detecting non-numeric key

Documentation Impact Milestone
☐ 2 ☐ 3 ☐ 4
☐ 6 ☐ 7 ☐ 8

Signature _____ Organization _____ Date _____

CORRECTION: (Brief description of work performed, including test cases used to confirm correction)

Solution: Adjust output character count for digit error 'schedule index output ESR to output the entire file.

Mod/Programs Changed: C1ESF

Hand Load ☐ Yes ☐ No

Work Performed by (Signature) _____ Date _____

CONFIRMATION: Corrections Verified by Product Assurance

Signature _____ Date _____

MTM No.(s) _____

Assigned to (Version ID) _____

WHITE - Originator Copy GREEN - Analyst CANARY - Originator Closed PINK - Product Control Closed GOLD - Product Control Open

Fig. 4. Omitted logic.

to configuration management practices. The decision to aw SPR's in this case was purely pragmatic—a large number of errors can be reviewed rapidly with minimal loss of enticity.

garding self-categorization, the study was built on the ise of exploring new ground. That is, to the author's vledge, no one has studied persistent software errors per se e. To avoid the possibility that traditional error categories were not appropriate to persistent errors, the traditional categories were deliberately avoided. Self-categorization, of course, has the flaw that the judgment of the individual researcher will have some impact on the final results. However, the traditional categories discussed in [10] are ambiguous enough that they share this problem.

In any case, 100 software errors from each of two projects were subjected to review at the raw SPR level via self-

TABLE I
PERSISTENT SOFTWARE ERRORS BY FREQUENCY OCCURRENCE (200 ERRORS EXAMINED, 100 ON EACH PROJECT)

Category:	Project A	Project B	Total:
1. Omitted logic (existing code too simple)	36	24	60
2. Failure to reset data	17	6	23
3. Regression error	5	12	17
4. Documentation in error (software correct)	10	6	16
5. Requirements inadequate	10	1	11
6. Patch in error	0	11	11
7. Commentary in error	0	11	11
8. If statement too simple	9	2	11
9. Referenced wrong data variable	6	4	10
10. Data alignment error (leftmost vs. rightmost bits, etc.)	4	3	7
11. Timing error caused data loss	3	3	6
12. Failure to initialize data	4	1	5
13. Other categories of lesser importance (total 4 or less) - logic too complex, compiler error, data storage overflow, expression incorrectly coded, pointer one off, dynamic allocation failure, data not included in checkpoint, microcode error, data boundary problem, macro error, multitasking synchronizing error, erroneous initialization, naming conventions violated, logic order incorrect, interface mismatch, data reset in error, parameter mismatch, inefficient code, data declaration wrong, bad overlay, statement label at wrong place, data clobbered.			

NOTE:
An error was allowed to tally in more than one category. "Failure to reset data", for example, is almost always a specific instance of "omitted logic." So are "if statement too simple" and "failure to initialize data." Any error could also be a "regression error."

categorization techniques. The errors were considered to be persistent on the basis that they were the most recent errors detected on those production-status projects. In most cases, this proved to be a sufficiently valid basis. In some cases, however, the errors were spawned by the correction of other persistent errors (these are commonly called "regression errors"). Errors of this class were included in the study on the grounds that such regression errors are just as costly as nonregression persistent errors; however, these errors were allowed to self-categorize a category for themselves.

An error was allowed to tally in more than one category. No attempt was made to provide mutually exclusive categories; the emphasis was on creating a realistic summary of the data as it was analyzed, and not to force it to fit an externally applied artifice. For example, the SPR in Fig. 2 would have been categorized both as "omitted logic" and "if statement too simple."

The process of analysis, then, was simply this: 1) project-specific, configuration-managed SPR forms, filed in chronological sequence, were examined one at a time; 2) using the "problem," "analysis," and "correction" information (see Figs. 2–4) the nature of the error was ascertained; 3) that error was categorized into its own and/or a previously selected category; 4) a tally was added to those categories. At the conclusion of analysis of a set of project-specific SPR's, tallies for the (variable number of) categories were summed. The categories for one project overlapped partially but not entirely with those of the other project (e.g., in Table I note that project A had no patching or commentary errors. Presumably project A did not allow patches and did not file SPR's on commentary errors).

THE FINDINGS

The findings of this study appear to be significant. That is, the categorized persistent SPR's show a consistent and definite pattern (see Table I).

The major finding of the study is that a large percentage of persistent software errors are instances of the software not being sufficiently complex to match the problem being solved. It is as if the programmer mind is straining to handle the complex interrelationships of a problem solution, and has failed. For example, a large number of such errors are the result of a predicate not having enough conditions—some flag or piece of data was not taken into account when it should have been—or of a variable not being reset to some baseline value after a major functional logic segment has finished dealing with it.

Here again, it is important to distinguish between the large problem and the small problem environment. Intuitively, it is easily seen that this kind of error is much more likely to emerge in the large rather than the small problem. The interrelationships between data and logic are much more entwined and complex in the large problem environment. It has even been said that "a 25 percent increase in problem complexity leads to a 100 percent increase in program complexity" [11]. And, in fact, most professional programmers have built software which they then realized was, in some areas, beyond their ability to comprehend (in the sense that its results were not predictable prior to its execution).

These problems of complexity may be considered to be design errors, and indeed many of them are. It is well known that design errors dominate the population of software errors (e.g., [4]). However, it must be recognized that they are the kind of design error which occurs at the most detailed level,

TABLE II
ERROR CATEGORY DEFINITION

Category:	Definition, Example:
1. Omitted logic	Code is lacking which should be present. Variable A is assigned a new value in logic path A but is not reset to the value required prior to entering path B.
2. Failure to reset data	Reassignment of needed value to a variable omitted. See example for "omitted logic."
3. Regression error	Attempt to correct one error causes another.
4. Documentation in error	Software and documentation conflict; software is correct. User manual says to input a value in inches, but program consistently assumes the value is in centimeters.
5. Requirements inadequate	Specification of the problem insufficient to define the desired solution. See Figure 4. If the requirements failed to note the interrelationship of the validity check and the disk schedule index, then this would also be a requirements error.
6. Patch in error	Temporary machine code change contains an error. Source code is correct, but "jump to 14000" should have been "jump to 14004."
7. Commentary in error	Source code comment is incorrect. Program says DO I=1,5 while comment says "loop 4 times."
8. IF statement too simple	Not all conditions necessary for an IF statement are present. IF A<B should be IF A<B AND B<C.
9. Referenced wrong data variable	Self-explanatory See Figure 3. The wrong queues were referenced.
10. Data alignment error	Data accessed is not the same as data desired due to using wrong set of bits. Leftmost instead of rightmost substring of bits used from a data structure.
11. Timing error causes data loss	Shared data changed by a process at an unexpected time. Parallel task B changes XYZ just before task A used it.
12. Failure to initialize data	Non-preset data is referenced before a value is assigned.

Lesser categories are not defined here.

where the design blends into its code. Since at this level it is not clear what is design and what is code, it would be simplistic to attach these errors to the broader category "design error."

WHAT TO DO ABOUT THE FINDINGS

The findings of this study are unsettling. They are unsettling not because they are a refocusing of our understandings of software problem solutions. They are unsettling because it is not at all clear what we should do about them.

Philosophically, what is needed is obvious. We need a human mind extender, one which makes it possible for the human mind to conceive problems and solutions beyond its current capacity. (Note that the analysis section of Fig. 2 says "insufficient brain power applied during design")!

Th[illegible]f course, is naive, given the current state-of-the-art. And [illegible], such a solution can at least be considered.

How could the mind be extended in those specific directions? Perhaps by very high-order languages, which remove solution details from the domain of the programmer into the domain of the compiler (analogous to computer hardware register management being moved into the high-order language compiler)?

Perhaps by a design aid which manages and analyzes design details which the human mind cannot?

Perhaps by a maintenance tool which extracts from existing software its underlying design elements, and subjects them to (human-assisted?) consistency analysis?

Those answers are not very satisfying, for they are beyond the state-of-the-computer-art. And yet they are promising, because they represent a level of computer application breakthrough which, if achieved, obviously transcends the software engineering problem which spawned it.

Of course, there are mundane but useful answers. If the designer, the implementer, and the tester employ a deep level of concentration and rigor, the omitted logic error is preventable or detectable. In-depth technical peer design reviews and peer code reviews can, for example, detect these errors before they become "persistent." Rigorous test case definition, especially where the test cases are driven by comprehensive specifications, can also detect most "persistent" errors early. Simple traditional desk checking, if properly applied, can also do the job.

The problem with these mundane but useful solutions is that, in the complex problems being solved by today's pro-

fessional programmer, the necessary concentration and rigor is difficult to achieve. Still, given proper application they can be viable solutions.

Conclusion

Persistent software errors are seen to be dominated by a class of error which can be categorized as "the failure of the problem solution to match the complexity of the problem to be solved." Examples of such errors are predicates with insufficient conditions, and failure to reset data to some baseline value after its use in a functional logic segment.

The solution to this class of problems is difficult. Somehow, the programmer's mind must be extended to encompass complexity beyond its current capability. This is obviously a solution beyond the current state-of-the-art.

Solutions which can be effective for today's large software system producer are maintaining awareness of the problem, and spending more time analyzing complex interrelationships via peer review, program desk checking, and rigorous testing. As is already well known, identifying those (persistent) problems early in the software life cycle can have major positive cost impact on total system cost.

Acknowledgment

The ideas and support of D. Feinberg, L. MacLaren, R. Noiseux, and E. Presson have been important in the development of this research.

References

[1] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[2] J. R. Brown, "Impact of MPP on system development," RADC-TR-77-121, 1977.
[3] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.
[4] A. B. Endres, "An analysis of errors and their causes in system programs," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 140–[illegible] June 1975.
[5] R. L. Glass, "Small versus large projects," in *Software Reliab. Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1979, pp. 21–25.
[6] E. Horowitz, *Practical Strategies for Developing Large Software Systems*. Reading, MA: Addison-Wesley, 1975.
[7] W. E. Howden, "An analysis of software validation techniques for scientific programs," Univ. Victoria Rep. DM-171-IR, 1979.
[8] J. R. Stanfield and A. M. Skrukrud, "Software acquisition management guidebook - Software maintenance volume," Syst. Develop. Corp., TM-5772/004/02, Nov. 1977.
[9] N. F. Schneidewind and H. M. Hoffman, "An experiment in software error data collection and analysis," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 276–286, May 1979.
[10] A. N. Sukert, "A multi-project comparison of software reliability models," in *Proc. AIAA Conf. Comput. in Aerospace*, 1977.
[11] S. N. Woodfield, "An experiment on unit increase in program complexity," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 76–79, Mar. 1979.

Robert L. Glass received the B.A. degree in mathematics from Culver-Stockton College, Canton, MO, in 1952, and the M.S. degree in mathematics from the University of Wisconsin in 1954.

He is presently a Senior Computing Specialist with Boeing Computer Services, a member of a team developing a UCSD Pascal based microcomputer system. Prior to that, he had spent 25 years in Aerospace computing—3 years with North American Aviation, 8 with Aerojet-General, and 14 with Boeing Aerospace. He has been an active participant in the Ada language and environment definition process, and in the JOVIAL language User's Group. He is experienced in scientific and commercial applications programming, and especially in the construction of system software. He is the author of three software engineering books—*Software Reliability Guidebook* (1979), *Software Maintenance Guidebook* (1981), and *Software Soliloquies* (1981)—and of four humorous computing books, two about computing projects which failed, and two about computing people. He has been published in several IEEE and ACM publications, and is a frequent contributor to *Computerworld* and *Datamation*. He has been an ACM National Lecturer for four years. His outlook is that of the experienced and skilled software craftsman, not that of a manager or an academician.

167

Principles of Program Design Induced from Experience with Small Public Programs

DOUGLAS COMER

Abstract—The art of programming is taught, learned, and often practiced as if programs are disposable, personal objects owned solely by the programmer. This paper uses examples to illustrate why real software is neither personal nor disposable; it shows how even simple programs are shared by others. From the examples, the paper extracts four principles for program development. Finally, it draws conclusions about programming practices and the education of programmers.

Index Terms—Programming, software design, software engineering.

Anything worth doing is worth doing well.
—anon

We do not know how to teach "good" programming habits.
—P. Wegner

I. Introduction

WHY is programmer[1] productivity low? Why do programmers ignore the work of others and build each program from scratch? These questions have prompted recent research into the specification, design, implementation, testing, and maintenance of production software (e.g., [5], [2], [8], [11]). Such research efforts, which are collectively referred to as "software engineering," aim to improve programmer productivity and increase the reliability, correctness, and cost effectiveness of the final product. Researchers have studied guidelines, techniques, and tools to aid in the development process. The varied approaches range from rigorous mathematical analysis and proof [6], [10] to management procedures [4], [1].

Despite research efforts to make software manufacturing a simple engineering process, it remains a complex art. Many projects still fall short of design goals, while others are delivered incredibly late. Many programmers cannot produce reliable products, and very few build on the work of others. More astonishingly, professionals who exhibit talent for producing useful, innovative systems seldom seem to know how they learned to do what others cannot. Usually, such successful individuals relate a series of battles they had with computer systems, and simply say that they learned a little from each.[2]

Battling computer systems extracts a heavy toll with little payoff. Programmers waste time and energy to discover a few simple facts. Many of the battles could be avoided altogether if the programmer adopted the correct attitude about programming and followed a few basic principles. The key to avoiding battles is sharing. Sharing each other's experiences and work increases programmer productivity and reduces frustration.

Before programmers can share each other's work, they must have confidence in it. Unfortunately, most programs are not designed for sharing; they contain hidden restrictions, dependencies, and flaws. This paper explores fundamental attitudes about programming and proposes changes that go beyond mere stylistic coding conventions. It develops four basic principles of programming, and asserts that following them is necessary for producing software that can be shared.

II. Private Versus Public Software

Brooks [3] claims that there are three types of programs: plain vanilla programs, program products, and systems products. Plain vanilla programs are what students write—they need not be reliable, portable, documented, or maintained. Most importantly, programmers think of writing vanilla programs as a *private* communication between the programmer and the machine; the program is thought of as a *personal* object owned solely by the programmer. The programmer is the specifier, designer, implementor, and end user rolled into one person. Program products, while not necessarily more complicated than vanilla programs, are *public*. They have to be correct, reliable, documented, and maintained because users who know nothing about the code depend on the output. Systems products are larger and more complicated than program products. Because of the sheer size and complexity, they cannot be designed, implemented, or maintained by a single individual. The production of a systems product involves cooperation and communication among a group of programmers, and is among the most challenging of human endeavors.

The process by which programmers are trained concentrates exclusively on tools and techniques for building small, vanilla programs rather than on guidelines and procedures for constructing and maintaining public software [12]. It instills the attitude that programs are personal property and that any program the author can use is sufficiently well designed. Beginners learn to write short, virtually useless programs that only serve to illustrate particular programming language constructs. Partly because of the limited time available in an academic course, advanced training fails to correct the misconcep-

Manuscript received April 11, 1980; revised July 28, 1980.

The author is with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

[1] From "programmer" will mean anyone who designs, implements, or maintains computer programs.

[2] In fact, [13] states that "people are taught how to code; programming is learned only by bitter experience."

time: one learns to write private programs which are discarded as soon as they run. The point here is that most young programmers emerge from their formal education with poor programming habits and attitudes ingrained. They assume that software is personal property, something that is not shared. When faced with a large systems product, they must adapt to both the fact that it will be public as well as the fact that it is large and complex.

There is a better way to teach programmers how to develop systems software than forcing them to jump from small, personal programs directly to complex public ones. A programmer should first face the task of maintaining small, public software. While experience with small programs cannot replace experience with large-scale systems, it can prepare the programmer to appreciate and understand the problem. Without such experience, programmers find the tools and techniques for large program construction meaningless. At our departmental computing facility, for example, each new programmer is given a small, public program as a first assignment. The programmer has a chance to learn what it means to maintain public software without jumping directly into maintenance of a large program like a compiler or operating system. More importantly, the programmers become conscious of others who depend on their work.

The next section presents two small, public programs which illustrate the differences between private and public software and show what a programmer can learn from working with a small program. The remaining sections list four principles of program development that have been extracted from experiences with small programs, and reviews them in light of the examples.

III. Experience with Small Programs

Using two small, public programs as examples, this section illustrates how software can be shared. The examples are: Lister, a program to paginate and format a file, and Grader, a program to maintain classroom grades.

Lister

Lister began as one of several utility programs written for use in a batch (remote job entry) environment. On the system in question, program source files and data files could be created interactively using a text editor. Up to 20 such files could be concatenated together and submitted as a batch job with a single command. The files were not otherwise accessible to a running program, however, so they could not be read or updated easily. To obtain a hardcopy listing of a file, one had to write a program that copied its input to its output, concatenate the program and the file together, and submit the result as a batch job.

The first version of Lister consisted of about 20 lines of PL/I code to print its input, inserting a page number and the date at the top of each page. To obtain a listing of files F1 and F2, one invoked Lister with an incantation like

```
?jw Lister,F1,F2
```

which would concatenate the Lister program file and the two files to be listed, and submit the result as a batch job. Because there were no listing utilities provided by the system, Lister was a convenient tool. In spite of the convenience, the program itself was trivial.

After some time, it became apparent that Lister lacked desirable abilities. Uppercase was the system standard for source files, but many files contained text with both upper- and lowercase. Because some printers handled only uppercase letters, the lowercase text was lost when output was routed to one of them. Line length posed another problem for Lister. Identification and sequence information in columns 73–80 of many files made listings of them difficult to read (the local text editor had the nasty habit of storing line numbers in columns 73–80 of each line unless the user told it not to do so). The second version of Lister compensated for these problems by translating all characters to uppercase and listing only the first 72 characters of a line. It recognized lines of the form %key=value as commands, however, to allow users to alter translation or change the line length. At any place in the input, the line

```
%T=NO
```

turned off translation, and the line

```
%T=YES
```

restored it. The command line

```
%C=100
```

changed the line length to 100, and the command line

```
%D=$
```

changed the delimiter for subsequent commands from "%" to "$." Commands were not listed with the rest of the input.

In addition to the program itself, the author maintained a document describing the use of Lister, including some examples. As others asked about the formatted listings, they were referred to the documentation. In a few months, the use of the Lister file began to increase, and shortly, there was a user population that depended on it.

Users began to suggest additions, changes, and improvements. Successive versions of Lister had more commands, and performed more formatting functions. By version 10, Lister allowed the user to change the page numbering; add headings; right justify text; direct output to a line printer, card punch, or another file; center text; underscore; change the page length; and include source files by name. The program had been rewritten from scratch at least three times in PL/I, SNOBOL4, and Assembler language (the latter was necessary to handle included files). The documentation had grown from 1 page to over 14 pages.

The users of Lister had changed, too. Instead of one user, there were many. Instead of one or two runs during the week, Lister was invoked every day, usually close to 20 times. Instead of listing existing files, users began to create files that only Lister could recognize and format. A dozen or more student term papers were prepared using Lister as the formatting program.

The change in use of Lister forced a change in the way it was maintained. Gradually, almost without effort, Lister had changed from a private program that could be changed at the

author's whim, into a public one. Since others depended on [illegible] remaining relatively stable, new versions had to be [illegible]ughly tested before they were installed. To test each new version, it was compiled and stored in the file Listerx. Those users who were anxious to try the new version invoked Listerx instead of Lister, and were helpful in testing as well as providing feedback on the design. After a period of testing, Listerx moved to Lister and the new version became the "production" version.

Grader

Like Lister, the Grader program began as a small, private program for use in computing classroom grades. The first version, written in SNOBOL4, supported a handful of commands to allow the user to enter student's names, identifiers, and grades; to compute weighted averages; and to print the results. Documentation for the program was contained in comments in the source file. When another professor asked about Grader, he was referred to the program source with the warning that he had better check the input carefully because the program did little to validate the data it received.

Despite the simplicity (and lack of adequate input validation), the popularity of Grader soared. Part of the popularity can be attributed to the lack of any reasonable competition: although many students and faculty claimed to have their own grading programs, none was documented or maintained. Part of the popularity, however, was due to commands which made [illegible]r convenient and flexible. For example, it accepted the [illegible]1 for "incomplete" and -2 for "omitted," and readjusted the specified weights on an individual basis to ignore the incomplete and omitted work when calculating a weighted average. At the end of the semester, instructors could change all incomplete grades to zero and recompute the weighted averages easily.

It became apparent that Grader needed to be rewritten and expanded, so version 2 was designed and implemented in Pascal. While the second version did support more commands, the major differences arose because it became a program product. Grader 2 checked the input carefully to find and report mistakes and nonsense. All input, including numbers, were read as characters; numeric input was converted to internal form only after it had been examined for errors. Grader also required the user to declare a maximum value for each homework, quiz, or examination score, and verified that individual scores fell in the correct range as they were entered.

As with Lister, users have suggested extensions and improvements to Grader over the past three years. Version 7 has 38 commands which allow one to change headings, footings, rearrange columns of output, omit highest or lowest grades in a group, add and drop students, print the grades, sort the listing, display scores as a bar chart, and even to write the grade matrix onto a file in a format convenient for input to other programs. The source file has grown from 300 lines to 2800 lines, [illegible]e user population has grown from one user to several dozen.

As a user population grew, changes to Grader had to be considered more seriously. The first version was used to assign grades to 35 students, and every calculation was verified with a calculator. Now, well over a thousand student grades are assigned each semester based on calculations performed by Grader (including complex adjustment of weights for individual students mentioned above). The procedures for modifying the current version (repairing problems) and for installing a new version (making user visible changes) are more complex. Files prepared as input to one version of Grader must be acceptable to later versions. Of course, new versions must be announced and documented before they replace the production version. Finally, public files are locked to prevent simultaneous access and update.

IV. Principles of Program Development

Lister and Grader come to mind as examples of the kind of program product from which programmers learn about public software before venturing into the area of large system engineering. While both programs were easily managed by one programmer, they have the essential ingredients that distinguish them from simple vanilla programs. From experiences with programs like Lister and Grader, several principles can be extracted. Following them leads a programmer toward a style and an attitude about programming that are amenable to work on larger, more complex systems.

The Principle of Use: Programs Will Be Used By Others

The principle of use sounds so simple that almost everyone agrees with it at first. Despite its simplicity, programmer training instills attitudes and habits that violate this principle. Most programmers assume that they will write plain programs unless they are told otherwise. Yet almost any program worth writing will be useful to others. Sooner or later the code will creep into public use. Private programs are the exception, not the rule. Keeping this in mind, programmers should make software reliable from the outset instead of beginning with a plain program and trying to add robustness as problems surface. They should assume all programs will be public and consider their reasons carefully before omitting reliability.

The experiences with Lister and Grader described above demonstrate how others appreciate and use even simple programs that are documented, correct, and reliable. It might be argued that Lister and Grader are atypical because they represent programs which provide general services of interest to many users; but they are not. Almost any program, however specialized, will be useful to someone besides the programmer who created it. For example, the author wrote a set of programs to enumerate a restricted class of trie index as part of his research. The programs were so specialized that it seemed obvious that no one, including the author, would ever use them again. In a surprising coincidence, a graduate student from another department needed a program to build trie indexes a few years later, and was able to lift code directly out of the original programs. Naturally, the program details had long been forgotten, so without comments in the programs to document the syntax of the input as well as the details of the algorithm, sharing would have been impossible.

Of course, not all software will become public—the term *fluffware* has been applied to one class of programs that are not used by anyone except their creators. Fluffware com-

prises those (usually quite trivial) programs that one pieces together rapidly, uses once, and then discards. For example, to find a pattern in a text file, one might devise a 5-line SNOBOL4 program and run it interactively without bothering to save the source program. Outside of fluffware, however, there is little that programmers keep entirely for themselves. Even small utility programs like Lister find their way around and eventually become public. One is forced to conclude that even though it may be simple or specialized, software that is worth keeping should be thought of as public (the public may consist of the programmer looking at the program long after it was written).

To follow the principle of use, one should do the following.

- Plan from the start to make programs public. This will save hours of debugging unreliable, incorrect programs when others start using them.
- Document a program or throw it away. This will save hours of explaining to others how to use programs (or reading code to find out yourself).
- Design software to be convenient for the uninitiated. This will save hours of interpreting the documentation.
- Label all output; save or print all input. This will save hours when users come to you for help, especially when they ask you to explain the output.

If programmers design, implement, and maintain all software as if it is production software, their documentation and programming habits improve. They begin to choose better names, comment code, and write more reliable programs. In the beginning, programming with others in mind takes time (Brooks [3] estimates that a program product requires three times the effort of a plain program). After a while, designing programs to be shared becomes habitual. One recognizes common pitfalls and problems and forms a set of standard solutions. Because programmers can depend on, and share each other's work, less time is wasted reinventing the wheel. In the long run, less time spent in repairing, explaining, and improving old programs leaves more time to devote to new ones. Unfortunately, only a few programmers take this principle seriously enough to benefit from making small programs correct, reliable, and documented.

The Principle of Misuse: Programs Will Be Abused

Users, sometimes the programmers themselves, supply the most unlikely input to programs. Empty files, binary files, very large files, object programs, letters in place of digits, digits in place of letters, excessively long lines, and other syntax errors are common. In addition, syntactically valid input will sometimes cause overflow, underflow, division by zero, table overflow, or subscript out of range problems. Finally, users can make subtle errors that cause unexpected output. For example, in Lister it was possible to specify that both a heading and page number should be printed at the same location on the page.

Another form of abuse occurs when users attempt to use a program for something other than what it was designed to do. For example, even though Grader provided only integer valued grades, one professor multiplied every grade by 1000 before entering it in order to obtain values accurate to three decimal places. Everything worked fine. The need for a bar chart to be printed. The program exceeded the output line limit trying to print a listing with over 100 000 lines (one line for each grade from 0 to 100 000).

If one takes the point of view of a user, proper techniques for handling the errors become clear. The program should respond with a reasonable message for any possible input, including an empty input file. Grader follows this precept by augmenting error messages with a listing of the input line and a pointer to the exact trouble spot. Naturally, error messages should be phrased so that a user who has never seen the source code can understand the problem; messages should never refer to variable names.

Error correction is more difficult and less important than error detection. If the program does attempt error correction, it should always call attention to amended or inserted input, or possibly incorrect (inaccurate) output that results. For example, an early version of Lister mistakenly truncated lines on the right edge of a page without telling the user, and an early version of Grader accepted fractional input but rounded to the nearest integer without telling the user. Both actions, which were intended to make the input more flexible, led to incorrect output. In both cases the program made the worst mistake possible by presenting output that looked reasonable, contained no warnings, but was incorrect. Later versions corrected these problems by informing users whenever input was altered.

To summarize, the programmer should do the following.

- Give a reasonable output for *any* input.
- Keep the program from terminating abnormally (even from printing a "dump").
- Call attention to corrected (altered) input or inaccurate output.

Of course, these rules must be tempered by the program specifications and the environment in which one works. In particular, the interpretation of "reasonable" varies greatly from system to system, and from individual to individual.

The Principle of Evolution: Programs Will Change

Inevitably, one will improve, extend, or modify all programs. This principle applies to the most trivial looking programs as well as complex ones. On one hand, errors may crop up when a program does not live up to the advertised specifications. On the other hand, the use and needs of even an error-free program change gradually over time. Features may be added to extend a program's capability, unused features may be deleted to make the implementation more efficient, or existing features may be modified. In anticipation of change, one should expend energy to make the code readable and modular.

Belady [2] calls the phenomenon of evolution "the law of continuing change" and explains some of the causes. Parnas [11] describes specific ways programmers can plan modules for ease of expansion and contraction, and Kernighan and Plauger [9] give detailed rules for programming style and documentation. Suffice it to say that modules should be designed to ease modification, and that the source code should be commented and uniformly styled.

Looking at this issue from a user's point of view, one can say that public software should not evolve without warning.

Rather, user visible changes should be collected together into numbered versions. The version numbers should appear in the source code, documentation, the object program, and on the output to provide a link between the running program and the source from which it came. Similarly, minor repairs and changes that do not affect users should be collected together into revisions. The convention of numbering programs as version *v.r*, where *v* is the version number and *r* is the revision number, works nicely. Beginning with version 0.0, those versions with numbers less than 1.0 can be used while the program is being written, making version 1.0 the first released version.

Users need to be warned of imminent version changes, and should have a reasonable assurance that the new version is tested. Users also need access to up-to-date documentation, especially when a new version has been installed.

In summary, programmers who plan for change will do the following.

- Make programs readable.
- Write modules to make extension and contraction easy.
- Collect user-visible changes into numbered versions.
- Collect repairs in numbered revisions.
- Keep the documentation current.

The Principle of Migration: Programs Will Move To New Machines.

Since the need for most software outlives the machine on which the software runs, one should expect that programs eventually run in a different environment than the one in which they are created. Interesting and useful programs usually migrate to new machines rapidly, sometimes without the owner's knowledge.

As hardware becomes less expensive, portability will become even more important. Unfortunately, many programmers still think of their task as that of instructing a machine. They take advantage of the nuances and quirks of a particular machine to gain efficiency or reduce the programming effort. To ensure portability, programmers must learn to avoid machine details instead of exploiting them; they must think of programs as problem solutions instead of instructions to a machine.

Once programmers accept the principle that programs will be transported to new environments, they work to do the following.

- Write programs to solve problems, not to instruct machines.
- Use standard programming language features.
- Isolate and comment all machine dependencies.

Lister and Grader contrast sharply in portability. Grader, written in Pascal, has moved to several machines and compilers without major effort. Lister, on the other hand, was coded in assembly language, so the result of the hours of work that went into writing and maintaining it were left behind when the author moved to a new computing environment. Of course, some of the ideas have been incorporated into a [illegible] formatting program (written in Pascal this time), and others are already provided as utilities in the new environment. The fact remains, however, that the original code for Lister would still be maintained and used if it had survived the transportation process.

V. Summary and Conclusions

This paper has discussed small program design, and has emphasized that programmers should not regard programs as private objects. This does not mean that all public programs arise from private ones that seep into the public domain. Quite the opposite is true: many public programs are conceived, designed, and implemented from the start with the entire development effort targeted to produce a public program. In order to build such programs well, programmers must be accustomed to public software as a way of life. Writing and maintaining small public programs provides a reasonable way to learn about public software without facing the problems of large or complex systems products.

In order to design programs for sharing, programmers must remain conscious of the following principles:

1) programs will be used by others;
2) programs will be abused;
3) programs will change;
4) programs will move to new machines;

and the specific guidelines for programmers that follow from them. Of course, these are not the only principles that one must follow. Experiences with Lister, Grader, and other production software do demonstrate, however, that successful programs have been built from them, and that programs can fail when they are not followed.

Programmer training cannot continue to enforce the notion that software production consists of writing small, useless programs. Young programmers must learn to deal with public, production software as a way of life. The habits and attitudes necessary for good software engineering must be cultivated early and reenforced often; they must pervade programmer training. In particular, relegating software engineering to a single college course (e.g., [7]) will not suffice if the remainder of the training does not mandate good practices. During the past few years, the software industry as well as universities have adopted the attitude that style and structure are important parts of program production. We must now act on the premise that plain vanilla programs are as unacceptable as poorly styled ones.

Acknowledgment

The author would like to thank W. Tichy and P. Denning for several suggestions.

References

[1] F. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 11, pp. 56–73, Jan. 1972.
[2] L. Belady and M. Lehman, "The characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 106–138.
[3] F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[4] B. Daly, "Management of software development," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 230–242, May 1977.
[5] F. DeRemer and H. Kron, "Programming-in-the-large vs. programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.
[6] E. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.
[7] J. Horning and D. Wortman, "Software hut: A computer program engineering project in the form of a game," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 325–330, July 1977.
[8] L. Ivie, "The programmer's workbench—A machine for software

development," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 746–753, Oct. 1977.
[9] B. Kernighan and P. Plauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1978.
[10] H. Mills, "How to write correct programs and know it," IBM Corp. Rep. FSC 73-5008, Gaithersburg, MD, 1973.
[11] D. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128–138, Mar. 1979.
[12] A. Ralston and M. Shaw, "Curriculum '78—Is computer science really that unmathematical?," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 67–70, Feb. 1980.
[13] J. Yohe, "An overview of programming practices," *Computing Surveys*, vol. 6, pp. 221–243, Dec. 1974.

Douglas Comer received the B.S. degree in physics and mathematics from Houghton College, Houghton, NY, in 1971, and the Ph.D. degree in computer science from Pennsylvania State University, University Park, in 1976.

He is currently an Assistant Professor in the Department of Computer Sciences at Purdue University, West Lafayette, IN, a position he has held since 1976. His research interests include algorithms for data storage and retrieval, programming languages, and software engineering.

Dr. Comer is a member of the Association for Computing Machinery and Sigma Xi.

173

Editorial

A Case for a Standard for Systems Engineering Methodology

ANDREW P. SAGE, FELLOW, IEEE

EDITOR

AN OPEN SET of procedures which provides the means for solving problems has become known as a *methodology*. The tools of system engineering: words, mathematics, and graphics, are the elements of systems engineering methodology and are also the elements of communication. The combination of a set of tools, a set of proposed activities for problem solution, and a set of relations among the tools and activities constitutes a methodology. The tools for systems engineering, or the content of systems engineering, consist of a variety of algorithms and concepts which enable various activities within systems engineering. Particular sets of relations among tools and activities, which constitute the framework for systems engineering, are of special importance since existence of and use of an appropriate systems engineering methodology could do much not only with respect to dealing with the many considerations, interrelations, and controversial value judgments associated with contemporary problems, but also could be of major value in organizing, teaching and communicating the value of the systems approach to policy and decisionmakers. Further, an appropriate methodology and framework for the systems approach should allow problem resolution at institutional and value levels as well as at the symptomatic level where so much of our effort today seems directed, or misdirected. Thus there exists strong motivation to examine questions concerning whether a "standard" framework for systems engineering is appropriate and more communicable than the spectrum of frameworks currently extant.

The thesis of this editorial is that this is not only possible but highly desirable. To develop this thesis further, let us examine several of the many frameworks for systems engineering that have been proposed.

Arthur D. Hall, III, in 1969, published a comprehensive three-dimensional framework, or morphology, for systems engineering in this TRANSACTIONS [1]. This three-dimensional morphology represented an extension of his earlier pioneering study of systems engineering methodology [2]. Fig. 1 is an adaptation of Hall's representation of the three main dimensions of systems engineering. The *time* dimension includes the gross sequences or phases that are characteristic of systems work and extends from the initial conception of an idea in the policy or programatic phase through systems retirement or phase out. The *logic* dimension deals with the steps that are carried out in each of the systems engineering phases. The *knowledge* dimension refers to specialized knowledge from the various professions and disciplines. Our discussions here will be specifically concerned with the activity plane of systems engineering which consists of the time and logic dimensions or steps of systems engineering.

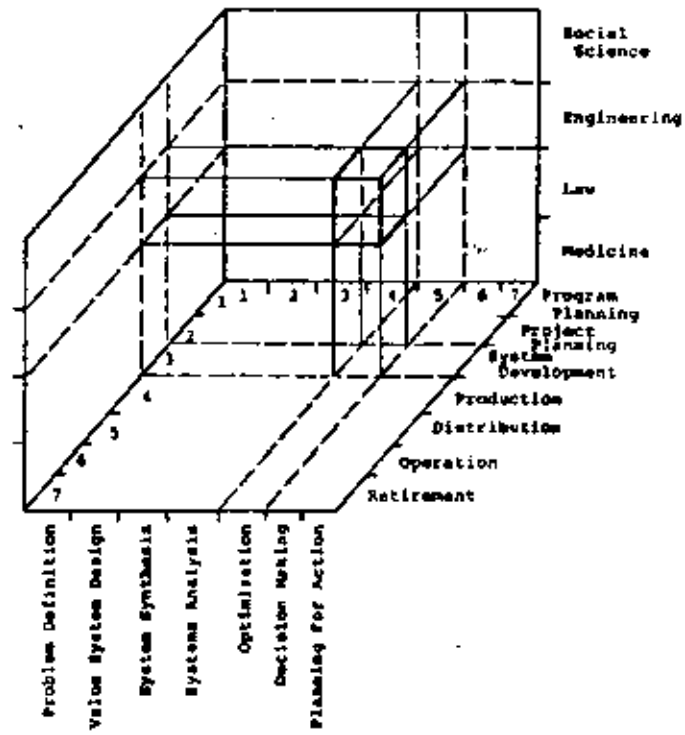


Fig. 1. Morphological box for systems engineering.

Programs or policies to be pursued, in the Hall activity matrix, are defined and selected in the *program planning* phase of systems engineering. The specific projects to be carried out are identified in the *project planning* phase of systems engineering. In *system development*, projects necessary to develop the system are implemented and production plans made. System elements as well as the total system are produced, and plans are made for installation of systems in the *production* phase of systems engineering. The system is installed and plans completed for system operation in the *installation* phase of systems engineering. In the *operation* phase of systems engineering, the system

serves its intended use. In the *retirement* phase of systems engineering, the system is withdrawn from use, and it is replaced by a new system or modified significantly.

The *problem definition* step of Hall's activity matrix for systems engineering is aimed at determining a descriptive scenario for the situation extant which presents as much history and data as needed to indicate how the particular problem under consideration came to be a problem. The purpose of *value system design* is to postulate and clarify objectives proposed for attainment to resolve problems identified in the previous step. Conceptualization of potential candidate policies, activities, controls, or whole systems which might allow attainment of objectives is the object of *systems synthesis*. The purpose of *systems analysis* (and modeling) is to develop insight into the interrelationships, behavior, and characteristics of proposed policies, activities, controls, or systems in terms of objective attainment and problem resolution and need satisfaction. In the *optimization* or ranking of alternatives phase, particular policy parameters and coefficients are selected such that each proposed policy is the best policy possible in terms of ultimate satisfaction of the value system. *Decisionmaking* is aimed at selecting one or more policies or systems worthy of further consideration. In *planning for action* (or to implement the next phase) we determine and reshape the previous six steps such that initiation of the next phase of a systems engineering effort can be begun propitiously. Each of these steps is iterative in nature, and refinements to the output of any given step can be made as a result of outputs obtained in succeeding steps. Hall indicates this very clearly in a cornucopia diagram of the systems process as well as a spiral diagram which indicates the many feedback loops inherent in an efficacious systems engineering framework.

Hill and Warfield, in 1972, published in this TRANSACTIONS [3] a definitive study of the program planning phase of the Hall activity matrix. Their extension consists not of a *modification* to the steps of the Hall activity matrix, but rather *further refinement* of the problem definition, value system design, and system synthesis steps, especially as they relate to program planning. Hill and Warfield suggest disaggregating problem definition into *needs assessment*, *alterables identification*, *constraints identification*, and determination of relevant *societal sectors*. They are explicitly concerned with a language with which to develop and portray group products in problem definition and select the language of graphics to fulfill this need. Self- and cross-interaction matrices are used to provide a unifying and satisfying visual picture of problem definition as well as value system design and system synthesis. They disaggregate the value system design step into two elements, namely, postulation of *objectives* and *objectives measures*. Further, they suggest use of cross- and self-interaction matrices to relate objectives and objectives measures to themselves as well as to the needs, constraints, and alterables from the problem definition step. In systems synthesis, Hill and Warfield propose *identifications of the policies activities or*

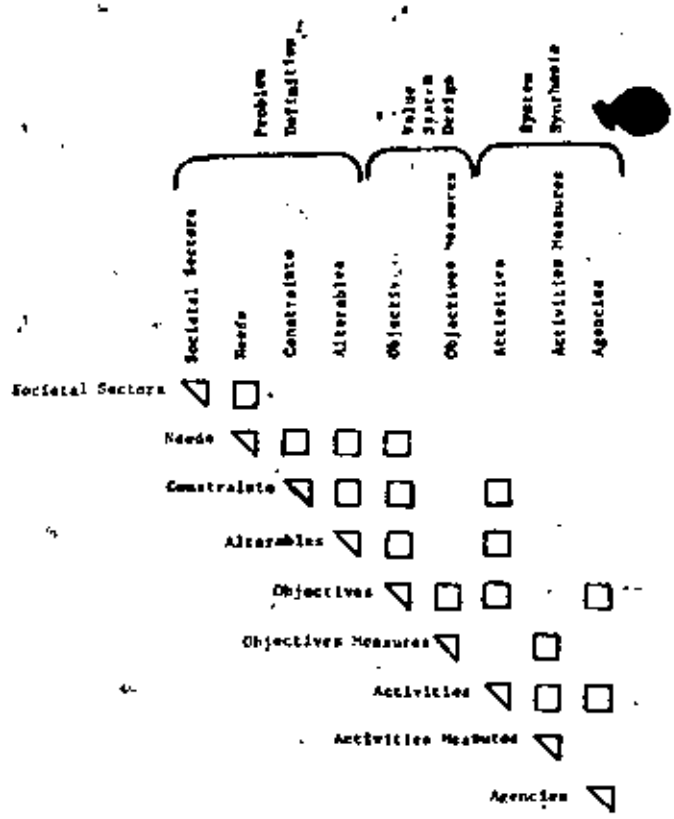


Fig. 2. Program planning linkages.

controls which may potentially satisfy objectives, *activities measures* with which to discern success (or lack thereof) in carrying out proposed policies and agencies to implement the activities. In this extension of the system synthesis step, major emphasis is given to the use of graphics to determine self- and cross-interaction matrices for the system synthesis elements of activities and activities measures, as well as the interrelationships between these elements and those of problem definition and value systems design. The complete end product of this effort is a graphic portrayal of the program planning linkages which consists of the problem definition elements (needs, constraints, alterables, and societal sectors), the value system design elements (objectives and objectives measures), and the system synthesis elements (agencies activities and activities measures). Fig. 2 illustrates graphically the program planning linkages and their relation to appropriate steps in the Hall activity matrix.

Reitman and Kiley [4] and *Reisman and de Kluyver* [5] have proposed, in 1973 and 1975, respectively, a two-dimensional framework for implementing system studies. This framework is in the form of a spiral diagram, readily convertible into a two-dimensional activity matrix, in which there are six steps, namely: *needs problem statement*, *value model*, *synthesis of solutions*, *analysis and/or test*, *evaluation*, and *decision*. Each of these activities steps are performed in each of six phases in the systems study. These phases are as follows: *feasibility study*, *preliminary design*, *detail design*, *pilot program or prototype*, *implementation construction and production*, *dissemination use and revision*, or *distribution consumption and modification*. Reisman, Kiley,

and de Kluyver suggest iteration of each of the steps of the logic dimension and ultimate convergence of the phases in the time dimension to a particular system configuration.

Gwilym Jenkins postulated, in a 1969 paper [6], another activity matrix for the systems engineering framework which consists of four phases, namely: *systems analysis, system design, implementation,* and *operation.* Within each of these four phases there are a number of steps. The steps vary in number and description. Within systems analysis, the steps are defined as *problem formulation, project organization, system definition, definition of a wider system, objectives of a wider system, objectives of the system, definition of the overall economic criterion,* and *information and data collection.* Within systems design, there exist five steps, namely: *forecasting, model building and simulation, optimization, control,* and *reliability.* Within implementation there are two steps, and they are *documentation* and *sanction approval, and construction.* Within operation there exist three steps: *initial operation, retrospective appraisal,* and *improved operation.* It is difficult to compare the system steps as delineated by Jenkins to those, for example, delineated by Hall (or others) in that Hall lists all steps for each phase and indicates that the amount of activity devoted to each step will necessarily be different depending upon the particular phase of the system study. Also, the particular set of tools used for a given step may vary considerably from one phase to the other. For example, system dynamics simulation might well be quite useful in the program planning phase of a systems study whereas discrete event digital simulation might be much less useful. However, after specific trial systems have been configured as in the operations phase, precisely the reverse situation may well be true. Thus it does not seem inappropriate to suggest that an activity matrix for systems engineering contain the same steps at each phase, as this allows for enhanced uniformity and clearly recognizes the possibility of different specific activities being utilized for a given step as a function of the phase. Also, it allows for differing intensities of effort associated with each step, as a function of the phases of the systems study.

Ralph Miles postulated, in a 1973 text [7], a framework for systems engineering consisting of five phases and six steps. The five phases are as follows: *system concept, system design, system implementation, system operation,* and *system completion.* The steps in the systems approach of Miles, presumably to be used in each of the phases, are *goal definition or problem statement, objectives and criteria development, systems synthesis, systems analysis, systems selection,* and *systems implementation.* Although each of these steps is not explicitly defined by Miles, their meaning seems fairly apparent from the above listing.

Yehezkel Dror is especially concerned with public policymaking in his 1968 text [8]. Thus Dror is not especially concerned with the phases of a systems engineering study other than the policy or program planning phase, the phase which leads to policymaking. Dror separates a complex policymaking problem into three phases, *metapolicymaking, policymaking,* and *postpolicymaking.* Metapolicy is a policy for making policy whereas postpolicymaking resembles the planning interface which must exist between policies or programs and projects for execution of policies. Within metapolicymaking Dror includes the following seven steps: *processing values, processing reality, processing problems, survey processing and developing resources, designing, evaluating, and redesigning the policymaking system, allocating problems values and resources,* and *determining policymaking strategy.* The policymaking subphase of policy and program planning also includes seven steps, namely: *suballocating resources, establishing operational goals, establishing the hierarchy of significant values, preparing a set of major alternative policies, determining predictions of significant benefits and costs of alternatives, optimization to identify best alternatives,* and *evaluating costs and benefits of the optimized alternatives.* The postpolicymaking subphase includes three steps. They are *motivating execution of policy, execution of policy,* and *evaluating the policymaking* which results after executing the policy. Dror mentions a final eighteenth step, *communication and feedback channels* to interconnect all steps. We shall not use this in our subsequent listing since each of the above delineated steps for this framework, as well as others, should be defined such as to include requisite communication and feedback channels. Dror is, like Hall, much concerned with the knowledge discipline. He details some twenty-three elements, professions, and subjects which can contribute to policy knowledge in the disciplines of knowledge.

John Gibson has proposed, in several papers [9]–[12], a six-step model for the logic structure or steps of systems engineering. In Gibson's listing of the fine structure of systems engineering, he includes problem definition, which is defined as developing a descriptive scenario to describe the situation extant and a normative scenario to list attributes of the solution as part of goal development. Gibson's steps, which are equivalent to the steps of the Hall activity matrix, and others examined here, are as follows: *goal development, establishment of criteria on which goal achievement can be judged, development of alternate candidate solutions, ranking of alternatives* by applying criteria previously established to the alternate solutions, *iteration of the steps* to obtain a deeper analysis of alternate solutions and to converge upon an acceptable solution, and *planning for action.* Gibson is more concerned with the initial planning and preliminary design phases for large systems studies such as designing a new city using systems engineering methodology, and does not detail the phases of a systems engineering framework or the professions. Since iteration seems to be an inherent feature of a well-planned systems study, as is the feedback and communications step of Dror, we shall delete this from our subsequent listing and assume its presence in all steps for all frameworks.

E. G. Hutchinson is, like many others, specifically concerned with the planning phase of systems engineering, in this particular case for urban transport systems [13]. In

Hutchinson's 1974 text [13], he delineates a five-step model of a planning process for systems engineering. These steps include: *problem definition, solution generation, solution analysis, evaluation and choice*, and *implementation*. In Hutchinson's description of the meaning of the steps he chooses for the systems engineering process, it is clear that problem definition is a major step as it includes determination of five related elements, namely: objectives, constraints, inputs, outputs, value functions, and decision criteria. Hutchinson acknowledges the relation of his five principal steps to the systems engineering methodology of Hall.

R. R. Baker, R. M. Michaels, and *E. S. Preston* list five steps for policy or program development in their 1975 text [14]. This work is explicitly concerned with public policy development for linking technical and political processes and does not consider the other phases of a systems engineering effort. The five steps in policy development are *perception of need, goal definition, policy analysis, alternative selection*, and *resource allocation*. Explicit definition of the above delineated steps are given in the work cited, but since the meaning of each of the steps is standard, these definitions will not be summarized here. Many of these steps are disaggregated into substeps in this work in much the same way as Hill and Warfield have disaggregated elements in the various program planning linkage steps.

E. S. Quade appears to make a distinction between systems methodology and systems approach in his 1975 text [15] and an earlier text [16] co-edited in 1968 with *W. I. Boucher*. Quade indicates that the systems approach is a concept or a way of looking at problems and is thus a practical philosophy for carrying out decision-oriented interdisciplinary research and not a method or technique or a fixed set of techniques. Quade does not appear to specifically delineate the phases of a systems engineering study. He lists five elements most important for analysis, including: objectives, alternatives, impacts, criteria, and models. Further, he delineates twelve specific steps which appear to represent the suggested steps in the fine structure of systems engineering. These twelve steps are as follows: *clarification of the problem, identification of objectives, measurement of effectiveness* (of various alternatives with respect to attainment of objectives), *determination of a criterion, determination of the environment, investigation of alternatives, formulation of models, data collection, comparison of alternatives, examination of analysis for sensitivities, consideration of deficiencies in the analysis*, and *summarization and recommendations*. Detailed descriptions of these various steps are provided in the cited works, but since definitions are standard and readily apparent, there appears little need to discuss them further here.

L. S. Hill, in 1970, published a paper concerning perspectives for systems engineering [17] in which he describes yet another activity matrix for systems engineering. Four phases appear in Hill's time dimension of systems engineering, namely: *analysis and planning, preliminary design, detailed design and test*, and *production design*. Within each of these four phases Hill indicates a number of steps. The steps differ somewhat from phase to phase although they are, in general, entirely commensurate with the steps of the Hall activity matrix, especially [redacted]. One recalls that emphasis will vary from phase to phase with respect to the intensity of effort devoted to the set of tasks which constitute activities within a given step. Within the analysis and planning phase Hill indicates six steps, and they are as follows: *exploration of problem and accumulation of information and data base, statement of the problem, identification of constraints and parameters, synthesis of a set of solutions, setting of objectives*, and *formulation of planning*. Preliminary design also consists of six steps, and these are as follows: *identification of building blocks, order of magnitude solutions, first-order solutions, preliminary layout sketches, synthesis of mathematical models and sensitivity analysis and testing*, and *evaluation and choice of preliminary design*. There are eight steps in the detailed design and test phase, namely: *evaluation planning organization and budgets, subsystems and component description, development of master layout configuration and functional drawings, detailed component and part design, determination of experimental models, testing of materials components models and prototypes, evaluation of test results*, and *assembly drawings and final design*. Within the fourth phase, production design, there are eight steps, and these include: *review and integration of component installations, evaluation for producibility and reliability, value analysis and quality assurance, completion and review of shop drawings, detailed production instructions and manufacturing and tooling, configuration control and coordination of engineering changes, issuance of training manuals*, and *control of issued drawings*. It would appear that the specific phrases used to describe the logic steps of the fine structure of his systems engineering model are, like the model of Jenkins, modifications of the seven general steps of Hall in terms more specifically applicable to the phases under consideration.

M. B. Kline and *M. W. Lifson* published, in 1971, a paper [18] in which they present an activity matrix for a systems engineering framework which appears roughly the transpose of the Hall activity matrix. The ten phases of the Kline–Lifson activity matrix are *concept formulation, system definition, preliminary design, engineering development, detail design, test and evaluation, production design, production and installation, operations and support*, and *modification and retirement*. Within each phase of activity there exist eight steps, namely: *gather available information, formulate value model, synthesize alternate solutions, analyze and/or test, evaluate, decide, optimize* (which apparently means iterate), and *communicate* (which apparently means plan for action).

There also exists a number of other models for systems engineering frameworks, many of which are known to the author and quite a number of which are undoubtedly unknown to the author. Nevertheless, our discussion indicated the existence of a large number of two- or three-dimensional morphologies for systems engineering. Figure 1 indicates the interrelationships of the steps of the fine structure of most of the models considered here with

[3] J. D. Hill and J. N. Warfield, "Unified program planning," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-2, no. 5, Nov. 1972, pp. 610-621.
[4] M. I. Taft and A. Reisman, *A General Approach to Problem Solving in Health Care Delivery Planning*, edited by A. Reisman and N. Kiley. New York: Gordon and Breach, 1973.
[5] A. Reisman and C. A. de Kluyver, *Strategies for Implementing System Studies* in *Implementing Operations Research-Management Science*, edited by R. L. Schultz and D. P. Slovin. New York: Elsevier North-Holland, 1975.
[6] G. M. Jenkins, "The systems approach," *J. Syst. Eng.*, vol. 1, no. 1, 1969.
[7] R. F. Miles, Jr., *Introduction to Systems Concepts* in *Systems Concepts*, edited by R. F. Miles, Jr. New York: Wiley, 1973.
[8] Y. Dror, *Public Policy Making Reexamined*. Scranton, PA: Chandler Publishing, 1968.
[9] J. E. Gibson, "A philosophy for urban simulations," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-2, no. 2, Apr. 1972, pp. 129-139.
[10] J. E. Gibson, "Why design a new city," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-3, no. 1, Jan. 1973, pp. 1-10.
[11] J. E. Gibson, "Eight scenarios for urban revitalization," *IEEE Proceedings Special Issue on Social Systems Engineering*, vol. 63, no. 3, Mar. 1975, pp. 444-451.
[12] J. E. Gibson, *On Designing the New City: A Systemic Approach*. New York: Wiley, 1977.
[13] G. B. Hutchinson, *Principles of Urban Transport Systems Planning*. New York: McGraw-Hill, 1974.
[14] R. F. Baker, R. M. Michaels, and E. S. Preston, *Public Policy Development: Linking the Technical and Political Processes*. New York: Wiley, 1975.
[15] E. S. Quade, *Analysis for Public Decisions*. New York: Elsevier North-Holland, 1975.
[16] E. S. Quade and W. I. Boucher, *Systems Analysis and Policy Planning: Applications and Defense*. New York: Elsevier North-Holland, 1968.
[17] L. S. Hill, "Systems engineering in perspective," *IEEE Trans. Eng. Manag.*, vol. EM-17, no. 4, Nov. 1970, pp. 124-131.
[18] M. B. Kline and M. W. Lifson, "Systems engineering and its application to the design of an engineering curriculum," *J. Syst. Eng.*, vol. 2, no. 1, Summer 1971, pp. 3-22.

177

TABLE I
COMPARISON OF THE STEPS FOR A SYSTEMS ENGINEERING FRAMEWORK AS PROPOSED BY VARIOUS AUTHORS

Authors of the Proposed Steps									
Hall [1]	Hall and Warfield* [3]		Reitman, Kiley and de Kluyver [4], [5]	Miles [7]	Gibson [9]–[12]	Hutchinson [13]	Baker, Michael's and Preston [14]	Quade and Boucher [15], [16]	Kline and Lifson [18]
Problem Definition	Problem definition	Needs Constraints Alternatives Societal sectors	Needs problem statement	Goal definition Problem statement	Goal development	Problem definition	Perception of need	Data collection Clarification of the problem Determination of the environment	Gather available information
Value System Design	Value system synthesis	Objectives Objectives Measures	Value model	Objectives and criteria development	Establish criteria for achievement of goals		Goal definition	Identification of objectives Determination of criterion	Formulate value model
System Synthesis	System synthesis	Activities or policies Controls or systems Activities measures Agencies	Synthesis of solutions	System synthesis	Development of alternative candidate solutions	Solution generation	Ø	Measurement of effectiveness of alternatives Investigation of alternatives	Synthesize alternate solutions
Systems Analysis	Systems analysis		Analysis and/or test	Systems analysis	Ø	Solution analysis	Policy analysis	Formulation of models Examination for sensitivities Consideration of deficiencies	Analyze and/or evaluate
Optimization	Optimization		Evaluation		Ranking of alternatives	Evaluation and choice	Alternative selection	Comparison of alternatives	Ø
Decision-making	Decisionmaking		Decision	System selection					Decide
Planning for Action	Planning for action		Ø	System implementation	Planning for action	Implementation	Resource allocation	Summarization and Recommendations	Communicate

* As noted in the text, Hall and Warfield acknowledge and utilize the Systems Engineering framework of Hall and expand upon those steps constituting the program planning linkage.

seven-step fine structure of Hall. For simplicity those models whose steps vary as a function of the phase are not shown. A similar table could be devised for the phases of systems engineering.

An essential feature of the systems engineering approach to wide-scope problem solution will generally be a team effort with an interdisciplinary group made up of people from a number of disciplines and professions. Such a diverse group will inevitably present communication problems due to a number of factors. A second and equally crucial source of difficulty in systems engineering practice is that much of the early conceptual work in a systems study is in the nature of policy and program concept manipulation and development. There exists an ever-present and considerable danger of much misunderstanding, apprehension, and delay in the early steps of a systems policy or program planning study. This is especially true relative to those steps concerned with problem definition, value system design, and system synthesis. Further, as Gibson has well documented in his system analysts decalogue [12], the client sponsoring a system study will often not fully understand the problem, and a major task of the systems engineering team is to help the client to gain this understanding. Often problems perceived by clients of a systems engineering team will be much too specific, and these specific problems must be embedded in more general questions and problems if a true understanding of the scenario extant is to be obtained. Further, it appears almost certain that many clients, as well as stakeholders, involved in potentially sponsoring or supporting a systems engineering study will not fully understand the systems engineering approach and may well be apprehensive or possibly antagonistic towards it. Each of these factors would appear to strongly suggest the high desirability of a systems engineering framework that can be readily communicated to an extraordinarily diverse group of people. It appears to this author that the many members of the existing multifaceted set of activity matrices and morphological boxes which constitute a framework for systems engineering may well appear so different to the non-systems-expert that communication is impeded. Equally damaging is that this may well convey the image of an unstructured, undisciplined, and uncoordinated disorderly fad whose use will generally not result in systematic application of the knowledge of science and technology to real-world tasks. A minor and more technical problem, albeit a serious one, with the situation extant may well be that the lack of a standardized framework and nomenclature for this framework makes it difficult to compare alternate systems designed from the viewpoint of different frameworks.

Assuming the truth of these premises, the conclusion can only be reached that much would be accomplished in the ultimate betterment of systems engineering professionals and our stakeholder public in general if an agreed-upon standard for a systems engineering framework and associated nomenclature and definitions could be achieved. Hopefully this editorial will stimulate and facilitate efforts in this direction. How might this be done? Will the benefits to our stakeholder public exceed the costs?

REFERENCES

[1] A. D. Hall, III, "Three-dimensional morphology of systems engineering," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-3, no. 2, Apr. 1969, pp. 156–160.
[2] A. D. Hall, III, *Methodology for Systems Engineering*. Princeton, N.J.: Van Nostrand, 1962.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTES AL TEMA I Y II

EL ENFOQUE ESTRUCTURADO DE DESARROLLO DE SOFTWARE

ANALISIS ESTRUCTURADO

Ing. Jorge I. Euan A.

ABRIL, 1982

Software Engineering

SOFTWARE ENGINEERING	BOEHM
STRUCTURED ANALYSIS FOR REQUIREMENTS DEFINITION	ROSS & SCHOMAN
PSL/PSA: A COMPUTER-AIDED TECHNIQUE FOR STRUCTURED DOCUMENTATION AND ANALYSIS OF INFORMATION PROCESSING SYSTEMS	TEICHROEW & HERSHEY
STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION	DEMARCO

1. Introduction

The annual cost of software in the U.S. is approximately 20 billion dollars. Its rate of growth is considerably greater than that of the economy in general. Compared to the cost of computer hardware, the cost of software is continuing to escalate along the lines predicted in Fig. 1 [1].* A recent SHARE study [2] indicates further that software demand over the years 1975-1985 will grow considerably faster (about 21-23 percent per year) than the growth rate in software supply at current estimated growth rates of the software labor force and its productivity per individual, which produce a combined growth rate of about 11.5-17 percent per year over the years 1975-1985.

In addition, as we continue to automate many of the processes which control our life-style — our medical equipment, air traffic control, defense system, personal records, bank accounts — we continue to trust more a more in the reliable functioning of this proliferating mass of software. *Software engineering* is the means by which we attempt to produce all of this software in a way that both cost-effective and reliable enough to deserve our trust. Clearly, it is a discipline which is important to establish well and to perform well.

---

*Another trend has been added to Fig.1: the growth of software maintenance, which will be discussed later.

This paper will begin with a definition of "software engineering." It will then survey the current state of the art of the discipline, and conclude with an assessment of likely future trends.

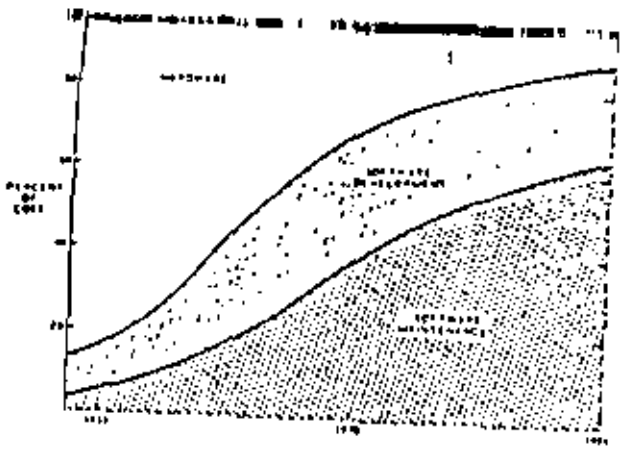


Figure 1. Hardware-software cost trends.

II. Definitions

Let us begin by defining "software engineering." We will define software to include not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs. By defining software in this broader sense, we wish to emphasize the necessity of considering the generation of timely documentation as an integral portion of the software development process. We can then combine this with a definition of "engineering" to produce the following definition.

> *Software Engineering:* The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Three main points should be made about this definition. The first concerns the necessity of considering a broad enough interpretation of the word "design" to cover the extremely important activity of software requirements engineering. The second point is that the definition should cover the entire software life cycle, thus including those activities of redesign and modification often termed "software maintenance." (Fig. 2 indicates the overall set of activities thus encompassed in the definition.) The final point is that our store of knowledge about software which can really be called "scientific knowledge" is a rather small base upon which to build an engineering discipline. But, of course, that is what makes software engineering such a fascinating challenge at this time.

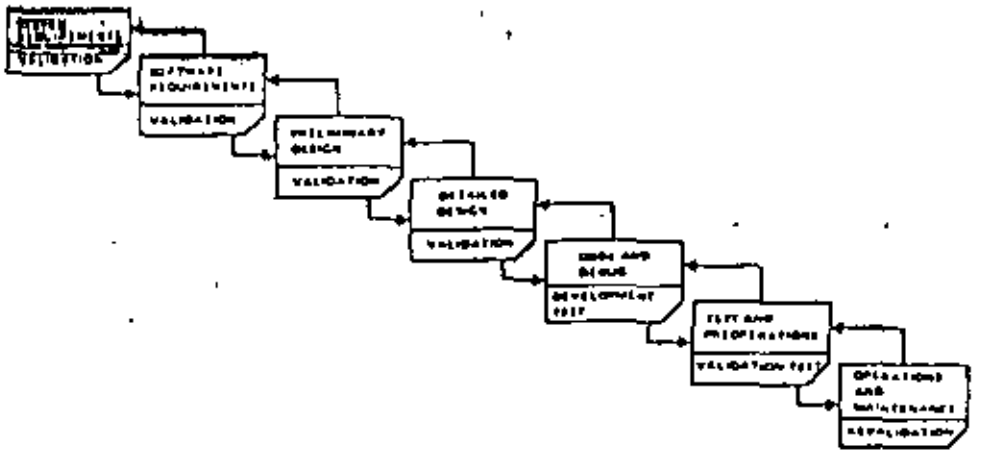


Figure 2. Software life cycle.

The remainder of this paper will discuss the state of the art of software engineering along the lines of the software life cycle depicted in Fig. 2. Section III contains a discussion of software requirements engineering, with some mention of the problem of determining overall system requirements. Section IV discusses both preliminary design and detailed design technology trends. Section V contains only a brief discussion of programming, as this topic is also covered in a companion article in this issue [3]. Section VI covers both software testing and the overall life cycle concern with software reliability. Section VII discusses the highly important but largely neglected area of software maintenance. Section VIII surveys software management concepts and techniques, and discusses the status and trends of integrated technology-management approaches to software development. Finally, Section IX concludes with an assessment of the current state of the art of software engineering with respect to the definition above.

Each section (sometimes after an introduction) contains a short summary of current practice in the area, followed by a survey of current frontier technology, and concluding with a short summary of likely trends in the area. The survey is oriented primarily toward discussing the domain of applicability of techniques (where and when they work) rather than how they work in detail. An extensive set of references is provided for readers wishing to pursue the latter.

III. Software requirements engineering

A. Crucial nature of software requirements engineering

Software requirements engineering is the discipline for developing a complete, consistent, unambiguous specification — which can serve as a basis for common agreement among all parties concerned — describing *what* the software product will do (but *not how* it will do it; this is to be done in the design specification).

The extreme importance of such a specification is only now becoming generally recognized. Its importance derives from two main characteristics: 1) it is easy to delay or avoid doing thoroughly; and 2) deficiencies in it are very difficult and expensive to correct later.

Fig. 3 shows a summary of current experience at IBM [4], GTE [5], and TRW on the relative cost of correcting software errors as a function of the phase in which they are corrected. Clearly, it pays off to invest effort in finding requirements errors early and correcting them in, say, 1 man-hour rather than waiting to find the error during operations and having to spend 100 man-hours correcting it.

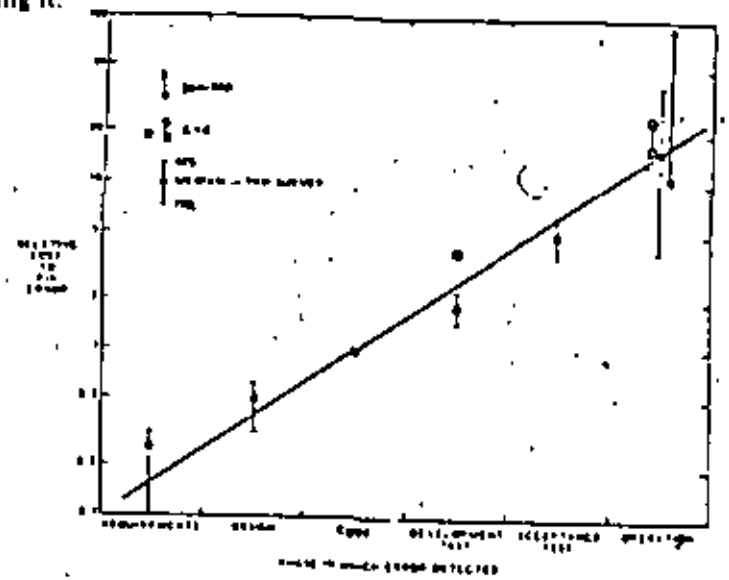


Figure 3. Software validation: the price of procrastination.

Besides the cost-to-fix problems, there are other critical problems stemming from a lack of a good requirements specification. These include [6]: 1) top-down design is impossible, for lack of a well-specified "top"; 2) testing is impossible because there is nothing to test against; 3) the user is frozen out, because there is no clear statement of what is being produced for him; and 4) management is not in control, as there is no clear statement of what the project team is producing.

B. Current practice

Currently, software requirements specifications (when they exist at all) are generally expressed in free-form English. They abound with ambiguous terms ("suitable," "sufficient," "real-time," "flexible") or precise-sounding terms with unspecified definitions ("optimum," "99.9 percent reliable") which are potential seeds of dissension or lawsuits once the software is produced. They have numerous errors; one recent study [7] indicated that the first independent review of a fairly good software requirements specification will find from one to four nontrivial errors per page.

The techniques used for determining software requirements are generally an ad hoc manual blend of systems analysis principles [8] and common sense. (These are the good ones; the poor ones are based on ad hoc manual blends of politics, preconceptions, and pure salesmanship.) Some formalized manual techniques have been used successfully for determining business system requirements, such as accurately defined systems (ADS), and time automated grid (TAG). The book edited by Couger and Knapp [9] has an excellent summary of such techniques.

C. Current frontier technology: Specification languages and systems

1) *ISDOS:* The pioneer system for machine-analyzable software requirements is the ISDOS system developed by Teichroew and his group at the University of Michigan [10]. It was primarily developed for business system applications, but much of the system and its concepts are applicable to other areas. It is the only system to have passed a market and operations test; several commercial, aerospace, and government organizations have paid for it and are successfully using it. The U.S. Air Force is currently using and sponsoring extensions to ISDOS under the Computer Aided Requirements Analysis (CARA) program.

ISDOS basically consists of a problem statement language (PSL) and a problem statement analyzer (PSA). PSL allows the analyst to specify his system in terms of formalized entities (INPUTS, OUTPUTS, REAL WORLD ENTITIES), classes (SETS, GROUPS), relationships (USES, UPDATES, GENERATES), and other information on timing, data volume, synonyms, attributes, etc. PSA operates on the PSL statements to produce a number of useful summaries, such as: formatted problem statements; directories and keyword indices; hierarchical structure reports; graphical summaries of flows and relationships; and statistical summaries. Some of these capabilities are actually more suited to supporting system design activities; this is often the mode in which ISDOS is used.

Many of the current limitations of ISDOS stem from its primary orientation toward business systems. It is currently difficult to express real-time performance requirements and man-machine interaction requirements, for example. Other capabilities are currently missing, such as support for configuration control, traceability to design and code, detailed consistency checking, and automatic simulation generation. Other limitations reflect deliberate, sensible design choices: the output graphics are crude, but they are produced in standard 8½ × 11 in size on any standard line printer. Much of the current work on ISDOS/CARA is oriented toward remedying such limitations, and extending the system to further support software design.

2) *SREP:* The most extensive and powerful system for software requirements specification in evidence today is that being developed under the Software Requirements Engineering Program (SREP) by TRW for the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC) [11, 12, 13]. Portions of this effort are derivative of ISDOS; it uses the ISDOS data management system, and is primarily organized into a language, the requirements statement language (RSL), and an analyzer, the requirements evaluation and validation system (REVS).

SREP contains a number of extensions and innovations which are needed for requirements engineering in real-time software development projects. In order to represent real-time performance requirements, the individual functional requirements can be joined into stimulus-response networks called R-Nets. In order to focus early attention on software testing and reliability, there are capabilities for designating "validation points" within the R-Nets. For early requirements validation, there are capabilities for automatic generation of functional simulators from the requirements statements. And, for adaptation to changing requirements, there are capabilities for configuration control, traceability to design, and extensive report generation and consistency checking.

Current SREP limitations again mostly reflect deliberate design decisions centered around the autonomous, highly real-time process-control problem of ballistic missile defense. Capabilities to represent large file processing and man-machine interactions are missing. Portability is a problem: although some parts run on several machines, other parts of the system run only on a TI-ASC computer with a very powerful but expensive multicolor interactive graphics terminal. However, the system has been designed with the use of compiler generators and extensibility features which should allow these limitations to be remedied.

3) *Automatic programming and other approaches:* Under the sponsorship of the Defense Advanced Research Projects Agency (DARPA), several researchers are attempting to develop "automatic programming" systems to replace the functions of [illegible] currently performed by programmers. If successful, could they drive [illegible] costs down to zero? Clearly not, because there would still be the need to determine what software the system should produce, i.e., the software requirements. Thus, the methods, or at least the forms, of capturing software requirements are of central concern in automatic programming research.

Two main directions are being taken in this research. One, exemplified by the work of Balzer at USC-ISI [14], is to work within a general problem context, relying on only general rules of information processing (items must be defined or received before they are used, an "if" should have both a "then" and an "else," etc.) to resolve ambiguities, deficiencies, or inconsistencies in the problem statement. This approach encounters formidable problems in natural language processing and may require further restrictions to make it tractable.

The other direction, exemplified by the work of Martin at MIT [15], is to work within a particular problem area, such as inventory control, where there is enough of a general model of software requirements and acceptable terminology to make the problems of resolving ambiguities, deficiencies, and inconsistencies reasonably tractable.

This second approach has, of course, been used in the past in various forms of "programming-by-questionnaire" and application generators [1, 2]. Perhaps the most widely used are the parameterized application generators developed for use on the IBM System/3. IBM has some more ambitious efforts on requirements specification underway, notably one called the Application Software Engineering Tool [16] and one called the Information Automat [17], but further information is needed to assess their current status and directions.

Another avenue involves the formalization and specification of required properties in a software specification (reliability, maintainability, portability, etc.). Some success has been experienced here for small-to-medium systems, using a "Requirements-Properties Matrix" to help analysts infer additional requirements implied by such considerations [18].

D. *Trends*

In the area of requirements statement languages, we will see further efforts either to extend the ISDOS-PSL and SREP-RSL capabilities to handle further areas of application, such as man-machine interactions, or to develop language variants specific to such areas. It is still an open question as to how general such a language can be and still retain its utility. Other open questions are those of the nature, "which representation scheme is best for describing requirements in a certain area?" BMDATC is sponsoring some work here in representing general data-processing system requirements for the BMD problem, involving Petri nets, state transition diagrams, and predicate calculus [11], but its outcome is still uncertain.

A good deal more can and will be done to extend the capability of requirements statement analyzers. Some extensions are fairly straightforward consistency checking; others, involving the use of relational operators to deduce derived requirements and the detection (and perhaps generation) of missing requirements are more difficult, tending toward the automatic programming work.

Other advances will involve the use of formal requirements statements to improve subsequent parts of the software life cycle. Examples include requirements-design-code consistency checking (one initial effort is underway), the automatic generation of test cases from requirements statements, and, of course, the advances in automatic programming involving the generation of code from requirements.

Progress will not necessarily be evolutionary, though. There is always a good chance of a breakthrough: some key concept which will simplify and formalize large regions of the problem space. Even then, though, there will always remain difficult regions which will require human insight and sensitivity to come up with an acceptable set of software requirements.

Another trend involves the impact of having formal, machine-analyzable requirements (and design) specifications on our overall inventory of software code. Besides improving software reliability, this will make our software much more portable; users will not be tied so much to a particular machine configuration. It is interesting to speculate on what impact this will have on hardware vendors in the future.

IV. Software design

A. *The requirements/design dilemma*

Ideally, one would like to have a complete, consistent, validated, unambiguous, machine-independent specification of software requirements before proceeding to software design. However, the requirements are not really validated until it is determined that the resulting system can be built for a reasonable *cost* — and to do so requires developing one or more software *designs* (and any associated hardware designs needed).

This dilemma is complicated by the huge number of degrees of freedom available to software/hardware system designers. In the 1950's, as indicated by Table 1, the designer had only a few alternatives to choose from in selecting a central processing unit (CPU), a set of peripherals, a programming language, and an ensemble of support software. In the 1970's, with rapidly evolving mini- and microcomputers, firmware, modems, smart terminals, data management systems, etc., the designer has an enormous number of alternative design components to sort out (possibilities) and to seriously choose from (likely choices). By the 1980's, the number of possible design combinations will be formidable.

Table 1
Design Degrees of Freedom for New Data Processing Systems (Rough Estimates)

Element	Choices (1950's)	Possibilities (1970's)	Likely Choices (1970's)
CPU	5	200	100
Op-Codes	fixed	variable	variable
Peripherals (per function)	1	200	100
Programming language	1	50	5–10
Operating system	0–1	10	5
Data management system	0	100	30

The following are some of the implications for the designer. 1) It is easier for him to do an outstanding design job. 2) It is easier for him to do a terrible design job. 3) He needs more powerful analysis tools to help him sort out the alternatives. 4) He has more opportunities for designing-to-cost. 5) He has more opportunities to design and develop tunable systems. 6) He needs a more flexible requirements-tracking and hardware procurement mechanism to support the above flexibility (particularly in government systems). 7) Any rational standardization (e.g., in programming languages) will be a big help to him, in that it reduces the number of alternatives he must consider.

B. *Current practice*

Software design is still almost completely a manual process. There is relatively little effort devoted to design validation and risk analysis before committing to a particular software design. Most software errors are made during the design phase. As seen in Fig. 4, which summarizes several software error analyses by IBM [4, 19] and TRW [20, 21], the ratio of design to coding errors generally exceeds 60:40. (For the TRW data, an error was called a design error if and only if the resulting fix required a change in the detailed design specification.)

Most software design is still done bottom-up, by developing software components before addressing interface and integration issues. There is, however, increasing successful use of top-down design. There is little organized knowledge of what a software designer does, how he does it, or of what makes a good software designer, although some initial work along these lines has been done by Freeman [22].

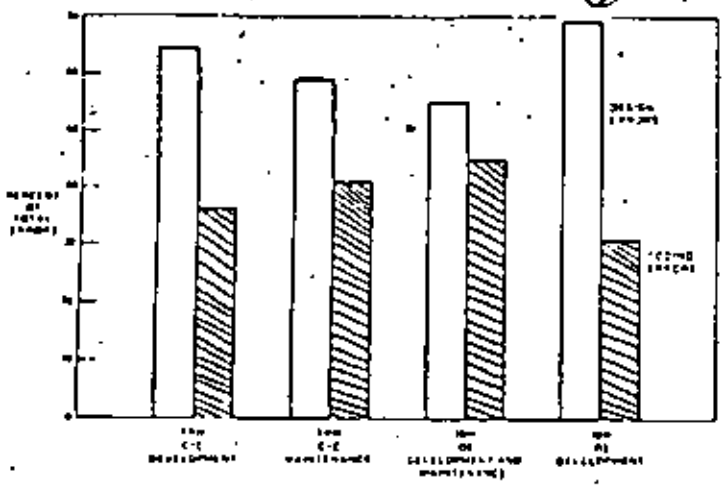


Figure 4. Most errors in large software systems are in early stages.

C. Current frontier technology

Relatively little is available to help the designer make the overall hardware-software tradeoff analyses and decisions to appropriately narrow the large number of design degrees of freedom available to him. At the micro level, some formalisms such as LOGOS [23] have been helpful, but at the macro level, not much is available beyond general system engineering techniques. Some help is provided via improved techniques for simulating information systems, such as the Extendable Computer System Simulator (ECSS) [24, 25], which make it possible to develop a fairly thorough functional simulation of the system for design analysis in a considerably shorter time than it takes to develop the complete design itself.

1) *Top-down design:* Most of the helpful new techniques for software design fall into the category of "top-down" approaches, where the "top" is already assumed to be a firm, fixed requirements specification and hardware architecture. Often, it is also assumed that the data structure has also been established. (These assumptions must in many cases be considered potential pitfalls in using such top-down techniques.)

What the top-down approach does well, though, is to provide a procedure for organizing and developing the control structure of a program in a way which focuses early attention on the critical issues of integration and interface definition. It begins with a top-level expression of a hierarchical control structure (often a top level "executive" routine controlling an "input," a "process," and an "output" routine) and proceeds to iteratively refine each successive lower-level component until the entire system is specified. The successive refinements, which may be considered as "levels of abstraction" or "virtual machines" [26], provide a number of advantages in improved understanding, communication, and verification of complex designs [27, 28]. In general, though, experience shows that some degree of early attention to bottom-level design issues is necessary on most projects [29].

The technology of top-down design has centered on two main issues. One involves establishing guidelines for *how to perform* successive refinements and to group functions into modules; the other involves techniques of *representing* the design of the control structure and its interaction with data.

2) *Modularization:* The techniques of structured design [30] (or composite design [31]) and the modularization guidelines of Parnas [32] provide the most detailed thinking and help in the area of module definition and refinement. Structured design establishes a number of successively stronger types of binding of functions into modules (coincidental, logical, classical, procedural, communicational, informational, and functional) and provides the guideline that a function should be grouped with those functions to which its binding is the strongest. Some designers are able to use this approach quite successfully; others find it useful for reviewing designs but not for formulating them; and others simply find it too ambiguous or complex to be of help. Further experience will be needed to determine how much of this is simply a learning curve effect. In general, Parnas' modularization criteria and guidelines are more straightforward and widely used than the levels-of-binding guidelines, although they may also be becoming more complicated as they address such issues as distribution of responsibility for erroneous inputs [33]. Along these lines, Draper Labs' Higher Order Software (HOS) methodology [34] has attempted to resolve such issues via a set of six axioms covering relations between modules and data, including responsibility for erroneous inputs. For example, Axiom 5 states, "Each module controls the rejection of invalid elements of its own, and only its own, input set."*

3) *Design representation:* Flow charts remain the main method currently used for design representation. They have a number of deficiencies, particularly in representing hierarchical control structures and data interactions. Also, their free-form nature makes it too easy to construct complicated, unstructured designs which are hard to understand and maintain. A number of representation schemes have been developed to avoid these deficiencies.

---

*Problems can arise, however, when one furnishes such a driven check with the power of an axiom. Suppose, for example, the input set contains a huge table or a master file. Is the module stuck with the job of checking it, by itself, every time?

The hierarchical input-process-output (HIPO) technique [35] represents software in a hierarchy of modules, each of which is represented by its inputs, its outputs, and a summary of the processing which connects the inputs and outputs. Advantages of the HIPO technique are its ease of use, ease of learning, easy-to-understand graphics, and disciplined structure. Some general disadvantages are the ambiguity of the control relationships (are successive lower level modules in sequence, in a loop, or in an if/else relationship?), the lack of summary information about data, the unwieldiness of the graphics on large systems, and the manual nature of the technique. Some attempts have been made to automate the representation and generation of HIPO's such as Univac's PROVAC System [36].

The structure charts used in structured design [30, 31] remedy some of these disadvantages, although they lose the advantage of representing the processes connecting the inputs with the outputs. In doing so, though, they provide a more compact summary of a module's inputs and outputs which is less unwieldy on large problems. They also provide some extra symbology to remove at least some of the sequence/loop/branch ambiguity of the control relationships.

Several other similar conventions have been developed [37, 38, 39], each with different strong points, but one main difficulty of any such manual system is the difficulty of keeping the design consistent and up-to-date, especially on large problems. Thus, a number of systems have been developed which store design information in machine-readable form. This simplifies updating (and reduces update errors) and facilitates generation of selective design summaries and simple consistency checking. Experience has shown that even a simple set of automated consistency checks can catch dozens of potential problems in a large design specification [21]. Systems of this nature that have been reported include the Newcastle TOPD system [40], TRW's DACC and DEVISE systems [21], Boeing's DECA system [41], and Univac's PROVAC [36]; several more are under development.

Another machine-processable design representation is provided by Caine, Farber, and Gordon's Program Design Language (PDL) System [42]. This system accepts constructs which have the form of hierarchical structured programs, but instead of the actual code, the designer can write some English text describing what the segment of code will do. (This representation was originally called "structured pidgin" by Mills [43].) The PDL system again makes updating much easier; it also provides a number of useful formatted summaries of the design information, although it still lacks some wished-for features to support terminology control and version control. The program-like representation makes it easy for programmers to read and write PDL, albeit less easy for nonprogrammers. Initial results in using the PDL system on projects have been quite favorable.

Once a good deal of design information is in machine-readable form there is a fair amount of pressure from users to do more with it: to generate core and time budgets, software cost estimates, first-cut data base descriptions etc. We should continue to see such added capabilities, and generally a further evolution toward computer-aided-design systems for software. Besides improvements in determining and representing control structures, we should see progress in the more difficult area of data structuring. Some initial attempts have been made by Hoare [44] and others to provide a data analog of the basic control structures in structured programming, but with less practical impact to date. Additionally, there will be more integration and traceability between the requirements specification, the design specification, and the code — again with significant implications regarding the improved portability of a user's software.

The proliferation of minicomputers and microcomputers will continue to complicate the designer's job. It is difficult enough to derive or use principles for partitioning software jobs on single machines; additional degrees of freedom and concurrency problems just make things so much harder. Here again, though, we should expect at least some initial guidelines for decomposing information processing jobs into separate concurrent processes.

It is still not clear, however, how much one can formalize the software design process. Surveys of software designers have indicated a wide variation in their design styles and approaches, and in their receptiveness to using formal design procedures. The key to good software design still lies in getting the best out of good people, and in structuring the job so that the less-good people can still make a positive contribution.

V. Programming

This section will be brief, because much of the material will be covered in the companion article by Wegner on "Computer Languages" [3].

A. Current practice

Many organizations are moving toward using structured code [28, 43] (hierarchical, block-oriented code with a limited number of control structures — generally SEQUENCE, IFTHENELSE, CASE, DOWHILE, and DOUNTIL — and rules for formatting and limiting module size). A great deal of terribly unstructured code is still being written, though, often in assembly language and particularly for the rapidly proliferating minicomputers and microcomputers.

B. Current frontier technology

(15)

Languages are becoming available which support structured code and additional valuable features such as data typing and type checking (e.g., Pascal [45]). Extensions such as concurrent Pascal [46] have been developed to support the programming of concurrent processes. Extensions to data typing involving more explicit binding of procedures and their data have been embodied in recent languages such as ALPHARD [47] and CLU [48]. Metacompiler and compiler writing system technology continues to improve, although much more slowly in the code generation area than in the syntax analysis area.

Automated aids include support systems for top-down structured programming such as the Program Support Library [49], Process Construction [50], TOPD [40], and COLUMBUS [51]. Another novel aid is the Code Auditor program [50] for automated standards compliance checking — which guarantees that the standards are more than just words. Good programming practices are now becoming codified into style handbooks, i.e., Kernighan and Plauger [52] and Ledgard [53].

C. Trends

It is difficult to clean up old programming languages or to introduce new ones into widespread practice. Perhaps the strongest hope in this direction is the current Department of Defense (DoD) effort to define requirements for its future higher order programming languages [54], which may eventually lead to the development and widespread use of a cleaner programming language. Another trend will be an increasing capability for automatically generating code from design specifications.

VI. Software testing and reliability

A. Current practice

Surprisingly often, software testing and reliability activities are still not considered until the code has been run the first time and found not to work. In general, the high cost of testing (still 40-50 percent of the development effort) is due to the high cost of reworking the code at this stage (see Fig. 3), and to the wasted effort resulting from the lack of an advance test plan to efficiently guide testing activities.

In addition, most testing is still a tedious manual process which is error-prone in itself. There are few effective criteria used for answering the question, "How much testing is enough?" except the usual "when the budget (or schedule) runs out." However, more and more organizations are now using disciplined test planning and some objective criteria such as "exercise every instruction" or "exercise every branch," often with the aid of automated test monitoring tools and test case planning aids. But other technologies, such as mathematical proof techniques, have barely begun to penetrate the world of production software.

(16)

B. Current frontier technology

1) *Software reliability models and phenomenology:* Initially, attempts to predict software reliability (the probability of future satisfactory operation of the software) were made by applying models derived from hardware reliability analysis and fitting them to observed software error rates [55]. These models worked at times, but often were unable to explain actual experienced error phenomena. This was primarily because of fundamental differences between software phenomenology and the hardware-oriented assumptions on which the models were based. For example, software components do not degrade due to wear or fatigue; no imperfection or variations are introduced in making additional copies of a piece of software (except possibly for a class of easy-to-check copying errors); repair of a software fault generally results in a different software configuration than previously, unlike most hardware replacement repairs.

Models are now being developed which provide explanations of the previous error histories in terms of appropriate software phenomenology. They are based on a view of a software program as a mapping from a space of inputs into a space of outputs [56], of program operation as the processing of a sequence of points in the input space, distributed according to an operational profile [57], and of testing as a sampling of points from the input space [56] (see Fig. 5). This approach encounters severe problems of scale on large programs, but can be used conceptually as a means of appropriately conditioning time-driven reliability models [58]. Still, we are a long way off from having truly reliable reliability-estimation methods for software.

Figure 5. Input space sampling provides a basis for software reliability measurement.

2) *Software error data:* Additional insights into reliability estimation have come from analyzing the increasing data base of software errors. For example, the fact that the distributions of serious software errors are dissimilar from the distributions of minor errors [59] means that we need to define "errors" very carefully when using reliability prediction models. Further, another study [60] found that the rates of fixing serious errors and of fixing minor errors vary with management direction. ("Close out all problems quickly" generally gets minor simple errors fixed very quickly, as compared to "Get the serious problems fixed first.")

– Other insights afforded by software data collection include better assessments of the relative efficacy of various software reliability techniques [4, 19, 60], identification of the requirements and design phases as key leverage points for cost savings by eliminating errors earlier (Figs. 2 and 3), and guidelines for organizing test efforts (for example, one recent analysis indicated that over half the errors were experienced when the software was handling data singularities and extreme points [60]). So far, however, the proliferation of definitions of various terms (error, design phase, logic error, validation test), still make it extremely difficult to compare error data from different sources. Some efforts to establish a unified software reliability data base and associated standards, terminology and data collection procedures are now under way at USAF Rome Air Development Center, and within the IEEE Technical Committee on Software Engineering.

3) *Automated aids:* Let us sketch the main steps of testing between the point the code has been written and the point it is pronounced acceptable for use, and describe for each step the main types of automated aids which have been found helpful. More detailed discussion of these aids can be found in the surveys by Reifer [61] and Ramamoorthy and Ho [62] which in turn have references to individual contributions to the field.

a) *Static code analysis:* Automated aids here include the usual compiler diagnostics, plus extensions involving more detailed data-type checking. Code auditors check for standards compliance, and can also perform various type-checking functions. Control flow and reachability analysis is done by structural analysis programs (flow charters have been used for some of the elementary checks here, "structurizers" can also be helpful). Other useful static analysis tools perform set-use analysis of data elements, singularity analysis, units consistency analysis, data base consistency checking, and data-versus-code consistency checking.

b) *Test case preparation:* Extensions to structural analysis programs provide assistance in choosing data values which will make the program execute along a desired path. Attempts have been made to automate the generation of such data values; they can generally succeed for simple cases, but run into difficulty in handling loops or

branching on complex calculated values (e.g., the results of numerical integration). Further, these programs only help generate the *inputs;* the tester must still calculate the expected outputs himself.

Another set of tools will automatically insert instrumentation to verify that a desired path has indeed been exercised in the test. A limited capability exists for automatically determining the minimum number of test cases required to exercise all the code. But, as yet, there is no tool which helps to determine the most appropriate sequence in which to run a series of tests.

c) *Test monitoring and output checking:* Capabilities have been developed and used for various kinds of dynamic data-type checking and assertion checking, and for timing and performance analysis. Test output post-processing aids include output comparators and exception report capabilities, and test-oriented data reduction and report generation packages.

d) *Fault isolation, debugging:* Besides the traditional tools — the core dump, the trace, the snapshot, and the breakpoint — several capabilities have been developed for interactive replay or backtracking of the program's execution. This is still a difficult area, and only a relatively few advanced concepts have proved generally useful.

e) *Retesting (once a presumed fix has been made):* Test data management systems (for the code, the input data, and the comparison output data) have been shown to be most valuable here, along with comparators to check for the differences in code, inputs, and outputs between the original and the modified program and test case. A promising experimental tool performs a comparative structure analysis of the original and modified code, and indicates which test cases need to be rerun.

f) *Integration of routines into systems:* In general, automated aids for this process are just larger scale versions of the test data management systems above. Some additional capabilities exist for interface consistency checking, e.g., on the length and form of parameter lists or data base references. Top-down development aids are also helpful in this regard.

g) *Stopping:* Some partial criteria for thoroughness of testing can and have been automatically monitored. Tools exist which keep a cumulative tally of the number or percent of the instructions or branches which have been exercised during the test program, and indicate to the tester what branch conditions must be satisfied in order to completely exercise all the code or branches. Of course, these are far from complete criteria for determining when to stop testing; the completeness question is the subject of the next section.

4) *Test sufficiency and program proving:* If a program's input space and output space are finite (where the input space includes not only all possible incoming inputs, but also all possible values in the program's data base), then one can construct a set of "black box" tests (one for each point in the input space) which can show conclusively that the program is correct (that its behavior matches its specification).

In general, though, a program's input space is infinite; for example, it must generally provide for rejecting unacceptable inputs. In this case, a finite set of black-box tests is not a sufficient demonstration of the program's correctness (since, for any input x, one must assure that the program does not wrongly treat it as a special case). Thus, the demonstration of correctness in this case involves some formal argument (e.g., a proof using induction) that the dynamic performance of the program indeed produces the static transformation of the input space indicated by the formal specification for the program. For finite portions of the input space, a successful exhaustive test of all cases can be considered as a satisfactory formal argument. Some good initial work in sorting out the conditions under which testing is equivalent to proof of a program's correctness has been done by Goodenough and Gerhart [63] and in a review of their work by Wegner [64].

5) *Symbolic execution:* An attractive intermediate step between program testing and proving is "symbolic execution," a manual or automated procedure which operates on symbolic inputs (e.g., variable names) to produce symbolic outputs. Separate cases are generated for different execution paths. If there are a finite number of such paths, symbolic execution can be used to demonstrate correctness, using a finite symbolic input space and output space. In general, though, one cannot guarantee a finite number of paths. Even so, symbolic execution can be quite valuable as an aid to either program testing or proving. Two fairly powerful automated systems for symbolic execution exist, the EFFIGY system [65] and the SELECT system [66].

6) *Program proving (program verification):* Program proving (increasingly referred to as program verification) involves expressing the program specifications as a logical proposition, expressing individual program execution statements as logical propositions, expressing program branching as an expansion into separate cases, and performing logical transformations on the propositions in a way which ends by demonstrating the equivalence of the program and its specification. Potentially infinite loops can be handled by inductive reasoning.

In general, nontrivial programs are very complicated and time-consuming to prove. In 1973, it was estimated that about one man-month of expert effort was required to prove 100 lines of code [67]. The largest program to be proved correct to date contained about 2000 statements [68]. Again, automation can help out on some of the complications. Some automated verification systems exist, notably those of Good *et al.* [69] and von Henke and Luckham [70]. In general, such systems do not work on programs in the more common languages such as Fortran or Cobol. They work in languages such as Pascal [45], which has (unlike Fortran or Cobol) an axiomatic definition [71] allowing clean expression of program statements as logical propositions. An excellent survey of program verification technology has been given by London [72].

Besides size and language limitations, there are other factors which limit the utility of program proving techniques. Computations on "real" variables involving truncation and roundoff errors are virtually impossible to analyze with adequate accuracy for most nontrivial programs. Programs with nonformalizable inputs (e.g., from a sensor where one has just a rough idea of its bias, signal-to-noise ratio, etc.) are impossible to handle. And, of course, programs can be proved to be consistent with a specification which is itself incorrect with respect to the system's proper functioning. Finally, there is no guarantee that the proof is correct or complete; in fact, many published "proofs" have subsequently been demonstrated to have holes in them [63].

It has been said and often repeated that "testing can be used to demonstrate the presence of errors but never their absence" [73]. Unfortunately, if we must define "errors" to include those incurred by the two limitations above (errors in specifications and errors in proofs), it must be admitted that "program proving can be used to demonstrate the presence of errors but never their absence."

7) *Fault-tolerance:* Programs do not have to be error-free to be reliable. If one could just detect erroneous computations as they occur and compensate for them, one could achieve reliable operation. This is the rationale behind schemes for fault-tolerant software. Unfortunately, both detection and compensation are formidable problems. Some progress has been made in the case of software detection and compensation for hardware errors; see, for example, the articles by Wulf [74] and Goldberg [75]. For software errors, Randell has formulated a concept of separately-programmed, alternate "recovery blocks" [76]. It appears attractive for parts of the error compensation activity, but it is still too early to tell how well it will handle the error detection problem, or what the price will be in program slowdown.

C. Trends

As we continue to collect and analyze more and more data on how, when, where, and why people make software errors, we will get added insights on how to avoid making such errors, how to organize our validation strategy and tactics (not only in testing but throughout the software life cycle), how to develop or evaluate new automated aids, and how to develop useful methods for predicting software reliability. Some automated aids, particularly for static code checking, and for some dynamic-type or assertion checking, will be integrated into future programming languages and compilers. We should see some added useful criteria and associated aids for test completeness, particularly along the lines of ex-

ercising "all data elements" in some appropriate way. Symbolic execution capabilities will probably make their way into automated aids for test case generation, monitoring, and perhaps retesting.

Continuing work into the theory of software testing should provide some refined concepts of test validity, reliability, and completeness, plus a better theoretical base for supporting hybrid test/proof methods of verifying programs. Program proving techniques and aids will become more powerful in the size and range of programs they handle, and hopefully easier to use and harder to misuse. But many of their basic limitations will remain, particularly those involving real variables and nonformalizable inputs.

Unfortunately, most of these helpful capabilities will be available only to people working in higher order languages. Much of the progress in test technology will be unavailable to the increasing number of people who find themselves spending more and more time testing assembly language software written for minicomputers and microcomputers with poor test support capabilities. Powerful cross-compiler capabilities on large host machines and microprogrammed diagnostic emulation capabilities [77] should provide these people some relief after a while, but a great deal of software testing will regress back to earlier generation "dark ages."

VII. Software maintenance

A. Scope of software maintenance

Software maintenance is an extremely important but highly neglected activity. Its importance is clear from Fig. 1: about 40 percent of the overall hardware-software dollar is going into software maintenance today, and this number is likely to grow to about 60 percent by 1985. It will continue to grow for a long time, as we continue to add to our inventory of code via development at a faster rate than we make code obsolete.

The figures above are only very approximate, because our only data so far are based on highly approximate definitions. It is hard to come up with an unexceptional definition of software maintenance. Here, we define it as "the process of modifying existing operational software while leaving its primary functions intact." It is useful to divide software maintenance into two categories: software *update*, which results in a changed functional specification for the software, and software *repair*, which leaves the functional specification intact. A good discussion of software repair is given in the paper by Swanson [78], who divides it into the subcategories of corrective maintenance (of processing, performance, or implementation failures), adaptive maintenance (to changes in the processing or data environment), and perfective maintenance (for enhancing performance or maintainability).

For either update or repair, three main functions are involved in software maintenance [79].

Understanding the existing software: This implies the need for good documentation, good traceability between requirements and code, and well-structured and well-formatted code.

Modifying the existing software: This implies the need for software, hardware, and data structures which are easy to expand and which minimize side effects of changes, plus easy-to-update documentation.

Revalidating the modified software: This implies the need for software structures which facilitate selective retest, and aids for making retest more thorough and efficient.

Following a short discussion of current practice in software maintenance, these three functions will be used below as a framework for discussing current frontier technology in software maintenance.

B. Current practice

As indicated in Fig. 6, probably about 70 percent of the overall cost of software is spent in software maintenance. A recent paper by Elshoff [80] indicates that the figure for General Motors is about 75 percent, and that GM is fairly typical of large business software activities. Daly [5] indicates that about 60 percent of GTE's 10-year life cycle costs for real-time software are devoted to maintenance. On two Air Force command and control software systems, the maintenance portions of the 10-year life cycle costs were about 67 and 72 percent. Often, maintenance is not done very efficiently. On one aircraft computer, software development costs were roughly $75/instruction, while maintenance costs ran as high as $4000/instruction [81].

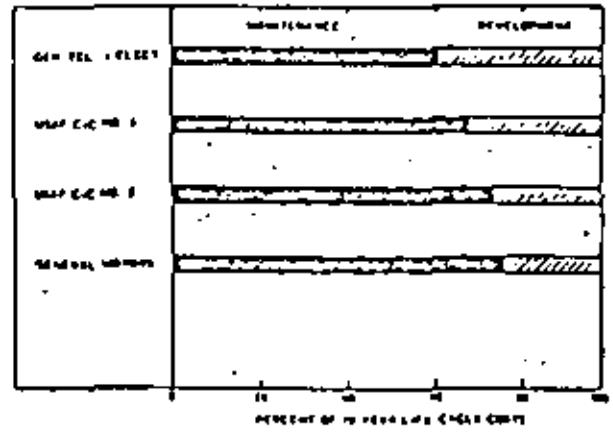


Figure 6. Software life-cycle cost breakdown.

Despite its size, software maintenance is a highly neglected activity. In general, less-qualified personnel are assigned to maintenance tasks. There are few good general principles and few studies of the process, most of them inconclusive.

Further, data processing practices are usually optimized around other criteria than maintenance efficiency. Optimizing around development cost and schedule criteria generally leads to compromises in documentation, testing, and structuring. Optimizing around hardware efficiency criteria generally leads to use of assembly language and skimping on hardware, both of which correlate strongly with increased software maintenance costs [1].

C. Current frontier technology

1) *Understanding the existing software:* Aids here have largely been discussed in previous sections: structured programming, automatic formatting, and code auditors for standards compliance checking to enhance code readability; machine-readable requirements and design languages with traceability support to and from the code. Several systems exist for automatically updating documentation by excerpting information from the revised code and comment cards.

2) *Modifying the existing software:* Some of Parnas' modularization guidelines [32] and the data abstractions of the CLU [48] and ALPHARD [47] languages make it easier to minimize the side effects of changes. There may be a maintenance price, however. In the past, some systems with highly coupled programs and associated data structures have had difficulties with data base updating. This may not be a problem with today's data dictionary capabilities, but the interactions have not yet been investigated. Other aids to modification are structured code, configuration management techniques, programming support libraries, and process construction systems.

3) *Revalidating the modified software:* Aids here were discussed earlier under testing; they include primarily test data management systems, comparator programs, and program structure analyzers with some limited capability for selective retest analysis.

4) *General aids:* On-line interactive systems help to remove one of the main bottlenecks involved in software maintenance: the long turnaround times for retesting. In addition, many of these systems are providing helpful capabilities for text editing and software module management. They will be discussed in more detail under "Management and Integrated Approaches" below. In general, a good deal more work has been done on the maintainability aspects of data bases and data structures than for program structures; a good survey of data base technology is given in a recent special issue of *ACM Computing Surveys* [82].

The increased concern with life cycle costs, particularly within the U.S. DoD [83], will focus a good deal more attention on software maintenance. More data collection and analysis on the growth dynamics of software systems, such as the Belady-Lehman studies of OS/360 [84], will begin to point out the high-leverage areas for improvement. Explicit mechanisms for confronting maintainability issues early in the development cycle, such as the requirements-properties matrix [18] and the design inspection [4] will be refined and used more extensively. In fact, we may evolve a more general concept of software quality assurance (currently focussed largely on reliability concerns), involving such activities as independent reviews of software requirements and design specifications by experts in software maintainability. Such activities will be enhanced considerably with the advent of more powerful capabilities for analyzing machine-readable requirements and design specifications. Finally, advances in automatic programming [14, 15] should reduce or eliminate some maintenance activity, at least in some problem domains.

VIII. Software management and integrated approaches

A. Current practice

There are more opportunities for improving software productivity and quality in the area of management than anywhere else. The difference between software project successes and failures has most often been traced to good or poor practices in software management. The biggest software management problems have generally been the following.

Poor Planning: Generally, this leads to large amounts of wasted effort and idle time because of tasks being unnecessarily performed, overdone, poorly synchronized, or poorly interfaced.

Poor Control: Even a good plan is useless when it is not kept up-to-date and used to manage the project.

Poor Resource Estimation: Without a firm idea of how much time and effort a task should take, the manager is in a poor position to exercise control.

Unsuitable Management Personnel: As a very general statement, software personnel tend to respond to problem situations as designers rather than as managers..

Poor Accountability Structure: Projects are generally organized and run with very diffuse delineation of responsibilities, thus exacerbating all the above problems..

Inappropriate Success Criteria: Minimizing development costs and schedules will generally yield a hard-to-maintain product. Emphasizing "percent coded" tends to get people coding early and to neglect such key activities as requirements and design validation, test planning, and draft user documentation.

Procrastination on Key Activities: This is especially prevalent when reinforced by inappropriate success criteria as above.

B. *Current frontier technology*

1) *Management guidelines:* There is no lack of useful material to guide software management. In general, it takes a book-length treatment to adequately cover the issues. A number of books on the subject are now available [85-95], but for various reasons they have not strongly influenced software management practice. Some of the books (e.g., Brooks [85] and the collections by Horowitz [86], Weinwurm [87], and Buxton, Naur, and Randell [88]) are collections of very good advice, ideas, and experiences, but are fragmentary and lacking in a consistent, integrated life cycle approach. Some of the books (e.g., Metzger [89], Shaw and Atkins [90], Hice *et al.* [91], Ridge and Johnson [92], and Gildersleeve [93]) are good on checklists and procedures but (except to some extent the latter two) are light on the human aspects of management, such as staffing, motivation, and conflict resolution. Weinberg [94] provides the most help on the human aspects, along with Brooks [85] and Aron [95], but in turn, these three books are light on checklists and procedures. (A second volume by Aron is intended to cover software group and project considerations.) None of the books have an adequate treatment of some items, largely because they are so poorly understood: chief among these items are software cost and resource estimation, and software maintenance.

In the area of software cost estimation, the paper by Wolverton [96] remains the most useful source of help. It is strongly based on the number of object instructions (modified by complexity, type of application, and novelty) as the determinant of software cost. This is a known weak spot, but not one for which an acceptable improvement has surfaced. One possible line of improvement might be along the "software physics" lines being investigated by Halstead [97] and others; some interesting initial results have been obtained here, but their utility for practical cost estimation remains to be demonstrated. A good review of the software cost estimation area is contained in [98].

2) *Management-technology decoupling:* Another difficulty of the above books is the degree to which they are decoupled from software technology. Except for the Horowitz and Aron books, they say relatively little about the use of such advanced-technology aids as formal, machine-readable requirements, top-down design approaches, structured programming, and automated aids to software testing.

Unfortunately, the management-technology decoupling works the other way, also: In the design area, for example, most treatments of top-down software design are presented as logical exercises independent of user or economic considerations. Most automated aids to software design provide little support for such management needs as configuration management, traceability to code or requirements, and resource estimation and control. Clearly, there needs to be a closer coupling between technology and management than this. Some current efforts to provide integrated management-technology approaches are presented next.

3) *Integrated approaches:* Several major integrated systems for software development are currently in operation or under development. In general, their objectives are similar: to achieve a significant boost in software development efficiency and quality through the synergism of a unified approach. Examples are the utility of having a complementary development approach (top-down, hierarchical) and set of programming standards (hierarchical, structured code); the ability to perform a software update and at the same time perform a set of timely, consistent project status updates (new version number of module, closure of software problem report, updated status logs); or simply the improvement in software system integration achieved when all participants are using the same development concept, ground rules, and support software.

The most familiar of the integrated approaches is the IBM "top-down structured programming with chief programmer teams" concept. A good short description of the concept is given by Baker [49]; an extensive treatment is available in a 15-volume series of reports done by IBM for the U.S. Army and Air Force [99]. The top-down structured approach was discussed earlier. The Chief Programmer Team centers around an individual (the Chief) who is responsible for designing, coding, and integrating the top-level control structure as well as the key components of the team's product; for managing and motivating the team personnel and personally reading and reviewing all their code; and also for performing traditional management and customer interface functions. The Chief is assisted by a Backup programmer who is prepared at anytime to take the Chief's place, a Librarian who handles job submission, configuration control, and project status accounting, and additional programmers and specialists as needed.

In general, the overall ensemble of techniques has been quite successful, but the Chief Programmer concept has had mixed results [99]. It is difficult to find individuals with enough energy and talent to perform all the above functions. If you find one, the project will do quite well; otherwise, you have concentrated most of the project risk in a single individual, without a good way of finding out whether or not he is in trouble. The Librarian and Programming Support Library concept have generally been quite useful, although the concept has been oriented toward a batch-processing development environment.

Another "structured" integrated approach has been developed and used at SofTech [38]. It is oriented largely around a hierarchical-decomposition design approach, guided by formalized sets of principles (modularity, abstraction, localization, hiding, uniformity, completeness, confirmability), processes (purpose, concept, mechanism, notation, usage), and goals (modularity, efficiency, reliability, understandability). Thus, it accommodates some economic considerations, although it says little about any other management considerations. It appears to work well for SofTech, but in general has not been widely assimilated elsewhere.

A more management-intensive integrated approach is the TRW software development methodology exemplified in the paper by Williams [50] and the TRW Software Development and Configuration Management Manual [100], which has been used as the basis for several recent government in-house software manuals. This approach features a coordinated set of high-level and detailed management objectives, associated automated aids — standards compliance checkers, test thoroughness checkers, process construction aids, reporting systems for cost, schedule, core and time budgets, problem identification and closure, etc. — and unified documentation and management devices such as the Unit Development Folder. Portions of the approach are still largely manual, although additional automation is underway, e.g., via the Requirements Statement Language [13].

The SDC Software Factory [101] is a highly ambitious attempt to automate and integrate software development technology. It consists of an interface control component, the Factory Access and Control Executive (FACE), which provides users access to various tools and data bases: a project planning and monitoring system, a software development data base and module management system, a top-down development support system, a set of test tools, etc. As the system is still undergoing development and preliminary evaluation, it is too early to tell what degree of success it will have.

Another factory-type approach is the System Design Laboratory (SDL) under development at the Naval Electronics Laboratory Center [102]. It currently consists primarily of a framework within which a wide range of aids to software development can be incorporated. The initial installment contains text editors, compilers, assemblers, and microprogrammed emulators. Later additions are envisioned to include design, development, and test aids, and such management aids as progress reporting, cost reporting and user profile analysis.

SDL itself is only a part of a more ambitious integrated approach, ARPA's National Software Works (NSW) [102]. The initial objective here has been to develop a "Works Manager" which will allow a software developer at a terminal to access a wide variety of software development tools on various computers available over the ARPANET. Thus, a developer might log into the NSW, obtain his source code from one computer, text-edit it on another, and perhaps continue to hand the program to additional computers for test instrumentation, compiling, executing, and postprocessing of output data. Currently, an initial version of the Works Manager is operational, along with a few tools, but it is too early to assess the likely outcome and payoffs of the project.

C. Trends

In the area of management techniques, we are probably entering a consolidation period, particularly as the U.S. DoD proceeds to implement the upgrades in its standards and procedures called for in the recent DoD Directive 5000.29 [104]. The resulting government-industry efforts should produce a set of software management guidelines which are more consistent and up-to-date with today's technology than the ones currently in use. It is likely that they will also be more comprehensible and less encumbered with DoD jargon; this will make them more useful to the software field in general.

Efforts to develop integrated, semiautomated systems for software development will continue at a healthy clip. They will run into a number of challenges which will probably take a few years to work out. Some are technical, such as the lack of a good technological base for data structuring aids, and the formidable problem of integrating complex software support tools. Some are economic and managerial, such as the problems of pricing services, providing tool warranties, and controlling the evolution of the system. Others are environmental, such as the proliferation of minicomputers and microcomputers, which will strain the capability of any support system to keep up-to-date.

Even if the various integrated systems do not achieve all their goals, there will be a number of major benefits from the effort. One is of course that a larger number of support tools will become available to a larger number of people (another major channel of tools will still continue to expand, though: the independent software products marketplace). More importantly, those systems which achieve a degree of conceptual integration (not just a free-form tool box) will eliminate a great deal of the semantic confusion which currently slows down our group efforts throughout the software life cycle. Where we have learned how to talk to each other about our software problems, we tend to do pretty well.

IX. Conclusions

Let us now assess the current state of the art of tools and techniques which are being used to solve software development problems, in terms of our original definition of software engineering: the practical application of *scientific knowledge* in the design and construction of software. Table II presents a summary assessment of the extent to which current software engineering techniques are based on solid scientific principles (versus empirical heuristics). The summary assessment covers four dimensions: the extent to which existing scientific principles apply across the entire software life cycle, across the entire

range of software applications, across the range of engineering-economic analyses required for software development, and across the range of personnel available to perform software development.

Table II
Applicability of Existing Scientific Principles

Dimension	Software Engineering	Hardware Engineering
Scope Across Life Cycle	Some principles for component construction and detailed design, virtually none for system design and integration, e.g., algorithms, automata theory.	Many principles applicable across life cycle, e.g., communication theory, control theory.
Scope Across Application	Some principles for "systems" software, virtually none for applications software, e.g., discrete mathematical structures.	Many principles applicable across entire application system, e.g., control theory application.
Engineering Economics	Very few principles which apply to system economics, e.g., algorithms.	Many principles apply well to system economics, e.g., strength of materials, optimization, and control theory.
Required Training	Very few principles formulated for consumption by technicians, e.g., structured code, basic math packages.	Many principles formulated for consumption by technicians, e.g., handbooks for structural design, stress testing, maintainability.

For perspective, a similar summary assessment is presented in Table II for hardware engineering. It is clear from Table II that software engineering is in a very primitive state as compared to hardware engineering, with respect to its range of scientific foundations. Those scientific principles available to support software engineering address problems in an area we shall call *Area 1: detailed design and coding of systems software* by *experts* in a relatively *economics-independent* context. Unfortunately, the most pressing software development problems are in an area we shall call *Area 2: requirements analysis design, test, and maintenance of applications software* by *technicians*[*] in an *economics-driven* context. And in Area 2, our scientific foundations are so slight that one can seriously question whether our current techniques deserve to be called "software engineering."

Hardware engineering clearly has available a better scientific foundation for addressing its counterpart of these Area 2 problems. This should not be too surprising, since "hardware science" has been pursued for a much longer time, is easier to experiment with, and does not have to explain the performance of human beings.

What is rather surprising, and a bit disappointing, is the reluctance of the computer science field to address itself to the more difficult and diffuse problems in Area 2, as compared with the more tractable Area 1 subjects of automata theory, parsing, computability, etc. Like most explorations into the relatively unknown, the risks of addressing Area 2 research problems in the requirements analysis, design, test and maintenance of applications software are relatively higher. But the prizes, in terms of payoff to practical software development and maintenance, are likely to be far more rewarding. In fact, as software engineering begins to solve its more difficult Area 2 problems, it will begin to lead the way toward solutions to the more difficult large-systems problems which continue to beset hardware engineering.

[*] For example, a recent survey of 74 installations in one large organization produced the following profile of its "average coder": 2 years college-level education, 3 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, "in over his head," and undermanaged. Given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve [illegible]h. This has strong implications for effective software engineering technology which, like effective software, [illegible] be well-matched to the people who must use it.

References

1. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5 (May 1973), pp. 48-59.
2. T.A. Dolotta, et al., *Data Processing in 1980-85* (New York: Wiley-Interscience, 1976).
3. P. Wegner, "Computer Languages," *IEEE Transactions on Computers*, Vol. C-25, No. 12 (December 1976), pp. 1207-25.
4. M.E. Fagan, *Design and Code Inspections and Process Control in the Development of Programs*, IBM Corporation, Report No. IBM-SDD TR-21.572 (December 1974).
5. E.B. Daly, "Management of Software Development," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 3 (May 1977), pp. 230-42.
6. W.W. Royce, "Software Requirements Analysis, Sizing, and Costing," *Practical Strategies for the Development of Large Scale Software*, ed. E. Horowitz (Reading, Mass.: Addison-Wesley, 1975).
7. T.E. Bell and T.A. Thayer, "Software Requirements: Are They a Problem?" *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976), pp. 61-68.
8. E.S. Quade, ed., *Analysis for Military Decisions* (Chicago: Rand-McNally, 1964).
9. J.D. Couger and R.W. Knapp, eds., *System Analysis Techniques* (New York: Wiley & Sons, 1974).
10. D. Teichroew and H. Sayani, "Automation of System Building," *Datamation*, Vol. 17, No. 16 (August 15, 1971), pp. 25-30.
11. C.G. Davis and C.R. Vick, "The Software Development System," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 69-84.
12. M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 60-69.
13. T.E. Bell, D.C. Bixler, and M.E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 49-60.
14. R.M. Balzer, *Imprecise Program Specification*, University of Southern California, Report No. ISI/RR-75-36 (Los Angeles: December 1975).
15. W.A. Martin and M. Bosyj, "Requirements Derivation in Automatic Programming," *Proceedings of the MRI Symposium on Computer Software Engineering* (April 1976).
16. N.P. Dooner and J.R. Lourie, *The Application Software Engineering Tool*, IBM Corporation, Research Report No. RC 5434 (Yorktown Heights, N.Y.: IBM Research Center, May 29, 1975).
17. M.L. Wilson, *The Information Automat Approach to Design and Implementation of Computer-Based Systems*, IBM Corporation (Gaithersburg, Md.: IBM Federal Systems Division, June 27, 1975).
18. B.W. Boehm, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," *Proceedings of the IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974), pp. 192-97.
19. A.B. Endres, "An Analysis of Errors and Their Causes in System Programs," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 140-49.
20. T.A. Thayer, "Understanding Software Through Analysis of Empirical Data," *AFIPS Proceedings of the 1975 National Computer Conference*, Vol. 44 (Montvale, N.J.: AFIPS Press, 1975), pp. 335-41.
21. B.W. Boehm, R.L. McClean, and D.B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 125-33.
22. P. Freeman, *Software Design Representation: Analysis and Improvements*, University of California, Technical Report No. 81 (Irvine, Calif.: May 1976).
23. E.L. Glaser, et al., "The LOGOS Project," *Proceedings of the IEEE Computer Conference* (1972), pp. 175-92.
24. N.R. Nielsen, *ECSS: Extendable Computer System Simulator*, Rand Corporation, Report No. RM-6132-PR/NASA (January 1970).
25. D.W. Kosy, *The ECSS II Language for Simulating Computer Systems*, Rand Corporation, Report No. R-1895-GSA (December 1975).
26. E.W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," *Software Engineering, Concepts and Techniques*, P. Naur, B. Randell, and J.N. Buxton, eds. (New York: Petrocelli/Charter, 1976).
27. H.D. Mills, *Mathematical Foundations for Structured Programming*, IBM Corporation, Report No. FSC 72-6012 (Gaithersburg, Md.: IBM Federal Systems Division, February 1972).

28. C.L. McGowan and J.R. Kelly, *Top-Down Structured Programming Techniques* (New York: Petrocelli/Charter, 1975).

29. B.W. Boehm, et al., "Structured Programming: A Quantitative Assessment," *Computer*, Vol. 8, No. 6 (June 1975), pp. 38-54.

30. W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2 (1974), pp. 115-39.

31. G.J. Myers, *Reliable Software Through Composite Design* (New York: Petrocelli/Charter, 1975).

32. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," *Communications of the ACM*, Vol. 15, No. 12 (December 1972), pp. 1053-58.

33. _____, "The Influence of Software Structure on Reliability," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 358-62.

34. M. Hamilton and S. Zeldin, "Higher Order Software — A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1 (March 1976), pp. 9-32.

35. *HIPO — A Design Aid and Documentation Technique*, IBM Corporation, Manual No. GC20-1851-0 (White Plains, N.Y.: IBM Data Processing Division, October 1974).

36. J. Mortison, "Tools and Techniques for Software Development Process Visibility and Control," *Proceedings of the ACM Computer Science Conference* (February 1976).

37. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices*, Vol. 8, No. 8 (August 1973), pp. 12-26.

38. D.T. Ross, J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process, Principles, and Goals," *Computer*, Vol. 8, No. 5 (May 1975), pp. 17-27.

39. M.A. Jackson, *Principles of Program Design* (New York: Academic Press, 1975).

40. P. Henderson and R.A. Snowden, "A Tool For Structured Program Development," *Proceedings of the 1974 IFIP Congress*, (Amsterdam, The Netherlands: North-Holland Publishing Co.), pp. 204-07.

41. L.C. Carpenter and L.L. Tripp, "Software Design Validation Tool," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 395-400.

42. S.H. Caine and E.K. Gordon, "PDL: A Tool for Software Design," *AFIPS Proceedings of the 1975 National Computer Conference*, Vol. 44 (Montvale, N.J.: AFIPS Press, 1975), pp. 271-76.

43. H.D. Mills, *Structured Programming in Large Systems*, IBM Corporation (Gaithersburg, Md.: IBM Federal Systems Division, November 1970).

44. C.A.R. Hoare, "Notes on Data Structuring," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).

45. N. Wirth, "An Assessment of the Programming Language Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 192-98.

46. P. Brinch-Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 199-208.

47. W.A. Wulf, *ALPHARD: Toward a Language to Support Structured Programs*, Carnegie-Mellon University, Internal Report (Pittsburgh, Pa.: April 30, 1974).

48. B.H. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4 (April 1974), pp. 50-59.

49. F.T. Baker, "Structured Programming in a Production Programming Environment," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 241-52.

50. R.D. Williams, "Managing the Development of Reliable Software," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 3-8.

51. J. Witt, "The COLUMBUS Approach," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4 (December 1975), pp. 358-63.

52. B.W. Kernighan and P.J. Plauger, *The Elements of Programming Style* (New York: McGraw-Hill, 1974).

53. H.F. Ledgard, *Programming Proverbs* (Rochelle Park, N.J.: Hayden, 1975).

54. W.A. Whitaker, et al., *Department of Defense Requirements for High Order Computer Programming Languages: 'Tinman,'* Defense Advanced Research Projects Agency (April 1976).

55. *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability* (April-May 1973).

56. E.C. Nelson, *A Statistical Basis for Software Reliability Assessment*, TRW Systems Group, Report No. TRW-SS-73-03 (Redondo Beach, Calif.: March 1973).

58. J.D. Musa, "Theory of Software Reliability and Its Application," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 3 (September 1975), pp. 312-27.

59. R.J. Rubey, J.A. Dana, and P.W. Biche, "Quantitative Aspects of Software Validation," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 150-55.

60. T.A. Thayer, M. Lipow, and E.C. Nelson, *Software Reliability Study*, TRW Systems Group, Report to RADC, Contract No. F30602-74-C-0036 (Redondo Beach, Calif.: March 1976).

61. D.J. Reifer, "Automated Aids for Reliable Software," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 131-42.

62. C.V. Ramamoorthy and S.B.F. Ho, "Testing Large Software With Automated Software Evaluation Systems," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 46-58.

63. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 156-73.

64. P. Wegner, "Report on the 1975 International Conference on Reliable Software," in *Findings and Recommendations of the Joint Logistics Commanders' Software Reliability Work Group*, Vol. II (November 1975), pp. 45-88.

65. J.C. King, "A New Approach to Program Testing," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 228-33.

66. R.S. Boyer, B. Elspas, and K.N. Levitt, "Select — A Formal System for Testing and Debugging Programs," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 234-45.

67. J. Goldberg, ed., *Proceedings of the Symposium on High Cost of Software*, Stanford Research Institute (Stanford, Calif.: September 1973), p. 63.

68. L.C. Ragland, "A Verified Program Verifier," Ph.D. Dissertation, University of Texas, 1973.

69. D.I. Good, R.L. London, and W.W. Bledsoe, "An Interactive Program Verification System," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1 (March 1975), pp. 59-67.

70. F.W. von Henke and D.C. Luckham, "A Methodology for Verifying Programs," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 156-64.

71. C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, Vol. 2 (1973), pp. 335-55.

72. R.L. London, "A View of Program Verification," *Proceedings of the 1975 International Conference on Reliable Software* (April 1975), pp. 534-45.

73. E.W. Dijkstra, "Notes on Structured Programming," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (New York: Academic Press, 1972).

74. W.A. Wulf, "Reliable Hardware-Software Architectures," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 233-40.

75. J. Goldberg, "New Problems in Fault-Tolerant Computing," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing* (Paris, France: June 1975), pp. 29-36.

76. B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 220-32.

77. R.K. McClean and B. Press, "Improved Techniques for Reliable Software Using Microprogrammed Diagnostic Emulation," *Proceedings of the IFAC Congress*, Vol. IV (August 1975).

78. E.B. Swanson, "The Dimensions of Maintenance," *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976).

79. B.W. Boehm, J.R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," *Proceedings of the IEEE/ACM Second International Conference on Software Engineering* (October 1976), pp. 592-605.

80. J.L. Elshoff, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2 (June 1976), pp. 113-20.

81. W.L. Trainor, "Software: From Satan to Saviour," *Proceedings of the NAECON* (May 1973).

82. E.H. Sibley, ed., *ACM Computing Surveys*, Vol. 8, No. 1 (March 1976). Special issue on data base management systems.

83. *Defense Management Journal*, Vol. II (October 1975). Special issue on software management.

84. L.A. Belady and M.M. Lehman, *The Evolution Dynamics of Large Programs*, IBM Corporation (Yorktown Heights, N.Y.: IBM Research Center, September 1975).

85. F.P. Brooks, *The Mythical Man-Month* (Reading, Mass.: Addison-Wesley, 1975).

86. E. I itz, ed., *Practical Strategies for Developing Large-Scale Software* (Reading, Mass.: Addison-Wesley, 1975).

87. G.F. Weinwurm, ed., *On the Management of Computer Programming* (New York: Auerbach, 1970).

88. P. Naur, B. Randell, and J.N. Buxton, eds., *Software Engineering, Concepts and Techniques, Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976).

89. P.J. Metzger, *Managing a Programming Project* (Englewood Cliffs, N.J.: Prentice-Hall, 1973).

90. J.C. Shaw and W. Atkins, *Managing Computer System Projects* (New York: McGraw-Hill, 1970)

91. G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology* (New York: American Elsevier, 1974).

92. W.J. Ridge and L.E. Johnson, *Effective Management of Computer Software* (Homewood, Ill.: Dow Jones-Irwin, 1973).

93. T.R. Gildersleeve, *Data Processing Project Management* (New York: Van Nostrand Reinhold, 1974).

94. G.M. Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Reinhold, 1971).

95. J.D. Aron, *The Program Development Process: The Individual Programmer* (Reading, Mass.: Addison-Wesley, 1974).

96. R.W. Wolverton, "The Cost of Developing Large-Scale Software," *IEEE Transactions on Computers*, Vol. C-23, No. 6 (June 1974).

97. M.H. Halstead, "Toward a Theoretical Basis for Estimating Programming Effort," *Proceedings of the ACM Conference* (October 1975), pp. 222-24.

98. *Summary Notes, Government/Industry Software Sizing and Costing Workshop*, USAF Electronic Systems Division (October 1974).

99. B.S. Barry and J.J. Naughton, *Chief Programmer Team Operations Description*, U.S. Air Force, Report No. RADC-TR-74-300, Vol. X (of 15-volume series), pp. 1-2-1-3.

100. *Software Development and Configuration Management Manual*, TRW Systems Group, Report No. TRW-SS-73-07 (Redondo Beach, Calif.: December 1973).

101. H. Bratman and T. Court, "The Software Factory," *Computer*, Vol. 8, No. 5 (May 1975), pp. 28-37.

102. *Systems Design Laboratory: Preliminary Design Report*, Naval Electronics Laboratory Center, Preliminary Working Paper TN-3145 (March 1976).

103. W.E. Carlson and S.D. Crocker, "The Impact of Networks on the Software Marketplace," *Proceedings of EASCON* (October 1974).

104. *Management of Computer Resources in Major Defense Systems*, U.S. Department of Defense, Directive 6000.29 (April 1976).

INTRODUCTION

The next article, by Ross and Schoman, is one of three papers chosen for inclusion in this book that deal with the subject of structured analysis. With its companion papers — by Teichroew and Hershey [Paper 23] and by DeMarco [Paper 24] — the paper gives a good idea of the direction that the software field probably will be following for the next several years.

The paper addresses the problems of traditional systems analysis, and anybody who has spent any time as a systems analyst in a large EDP organization immediately will understand the problems and weaknesses of "requirements definition" that Ross and Schoman relate — clearly *not* the sort of problems upon which academicians like Dijkstra, Wirth, Knuth, and most other authors in this book have focused! To stress the importance of proper requirements definition, Ross and Schoman state that "even the best structured programming code will not help if the programmer has been told to solve the wrong problem, or, worse yet, has been given a correct description, but has not understood it."

In their paper, the authors summarize the problems associated with conventional systems analysis, and describe the steps that a "good" analysis approach should include. They advise that the analyst separate his *logical*, or *functional*, description of the system from the *physical* form that it eventually will take; this is difficult for many analysts to do, since they assume, a priori, that the physical implementation of the system will consist of a computer.

Ross and Schoman also emphasize the need to achieve a consensus among typically disparate parties: the user liaison personnel who interface with the developers, the "professional" systems analyst, and management. Since all of these people have different interests and different viewpoints, it becomes all the more important that they have a common frame of reference — a common way of *modeling* the system-to-be. For this need, Ross and Schoman propose their solution: a proprietary package, known as SADT, that was developed by the consulting firm of SofTech for which the authors w k."

The SADT approach utilizes a top-down, partitioned, graphic model of a system. The model is presented in a logical, or abstract, fashion that allows for eventual implementation as a manual system, a computer system, or a mixture of both. This emphasis on graphic models of a system is distinctly different from that of the Teichroew and Hershey paper. It is distinctly similar to the approach suggested by DeMarco in "Structured Analysis and System Specification," the final paper in this collection. The primary difference between DeMarco and Ross/Schoman is that DeMarco and his colleagues at YOURDON inc. prefer circles, or "bubbles," whereas the SofTech group prefers rectangles.

Ross and Schoman point out that their graphic modeling approach can be tied in with an "automated documentation" approach of the sort described by Teichroew and Hershey. Indeed, this approach gradually is beginning to be adopted by large EDP organizations; but for installations that can't afford the overhead of a computerized, automated systems analysis package, Ross and Schoman neglect one important aspect of systems modeling. That is the "data dictionary," in which *all* of the data elements pertinent to the new system are defined *in the same logical top-down fashion as the rest of the model.* There also is a need to formalize mini-specifications, or "mini-specs" as DeMarco calls them; that is, the "business policy" associated with each bottom-level functional process of the system must be described in a manner far more rigorous than currently is being done.

A weakness of the Ross/Schoman paper is its lack of detail about problem solutions: More than half the paper is devoted to a description of the problems of conventional analysis, but the SADT package is described in rather sketchy detail. There are additional documents on SADT available from SofTech, but the reader still will be left with the fervent desire that Messrs. Ross and Schoman and their colleagues at SofTech eventually will sit down and put their ideas into a full-scale book.

Structured Analysis for Requirements Definition

I. The problem

The obvious assertion that "a problem unstated is a problem unsolved" seems to have escaped many builders of large computer application systems. All too often, design and implementation begin before the real needs and system functions are fully known. The results are skyrocketing costs, missed schedules, waste and duplication, disgruntled users, and an endless series of patches and repairs euphemistically called "system maintenance." Compared to other phases of system development, these symptoms reflect, by a large margin, the lack of an adequate approach to requirements definition.

Given the wide range of computer hardware now available and the emergence of software engineering as a discipline, most problems in system development are becoming less traceable to either the machinery or the programming [1]. Methods for handling the hardware and software components of systems are highly sophisticated, but address only part of the job. For example, even the best structured programming code will not help if the programmer has been told to solve the wrong problem, or, worse yet, has been given a correct description, but has not understood it. The results of requirements definition must be both complete and understandable.

In efforts to deal with these needs, the expressions "system architecture," "system design," "system analysis," and "system engineering" seem to be accepted terminology. But in truth, there is no widely practiced methodology for systems work that has the clarity and discipline of the more classical techniques used in construction and manufacturing enterprises. In manufacturing, for example, a succession of blueprints, drawings, and specifications captures all of the relevant requirements for a product. This complete problem definition and implementation plan allows the product to be made almost routinely, by "business as usual," with no surprises. In a good manufacturing operation, major troubles are avoided because even the first production run does not create an item for the first time. The item was created and the steps of forming and assembly were done mentally, in the minds of designers and engineers, long before the set of blueprints and specifications ever arrived at the production shop. That simulation is made possible only because the notations and discipline of the blueprinting methodology are so complete and so consistent with the desired item that its abstract representation contains all the information needed for its imaginary preconstruction.

Software system designers attempt to do the same of course, but being faced with greater complexity and less exacting methods, their successes form the surprises, rather than their failures!

Experience has taught us that system problems are complex and ill-defined. The complexity of large systems is an inherent fact of life with which one must cope. Faulty definition, however, is an artifact of inadequate methods. It can be eliminated by the introduction of well-thought-out techniques and means of expression. That is the subject of this paper. Systems can be manufactured, like other things, if the right approach is used. That approach must start at the beginning.

Requirements definition

Requirements definition includes, but is not limited to, the problem analysis that yields a functional specification. It is much more than that. Requirements definition must encompass everything necessary to lay the groundwork for subsequent stages in system development (Fig. 1). Within the total process, which consists largely of steps in a solution to a problem, only once is the problem itself stated and the solution justified — in requirements definition.

Requirements definition is a careful assessment of the needs that a system is to fulfill. It must say *why* a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say *what* system features will serve and satisfy this context. And it must say *how* the system is to be constructed. Thus, requirements definition must deal with three subjects.

1) *Context analysis:* The reasons *why* the system is to be created and why certain technical, operational, and economic feasibilities are the criteria which form *boundary conditions* for the system.

2) *Functional specification:* A description of *what* the system is to be, in terms of the functions it must accomplish. Since this is part of the problem statement, it must only present *boundary conditions* for considerations later to be taken up in system design.

3) *Design constraints:* A summary of conditions specifying *how* the required system is to be constructed and implemented. This does not necessarily specify which things will be in the system. Rather it identifies *boundary conditions* by which those things may later be selected or created.

Each of these subjects must be fully documented during requirements definition. But note that these are subjects, not documents. The contents of resulting documents will vary according to the needs of the development organization. In any case, they must be reference documents which justify all aspects of the required system, not design or detailed specification documents.

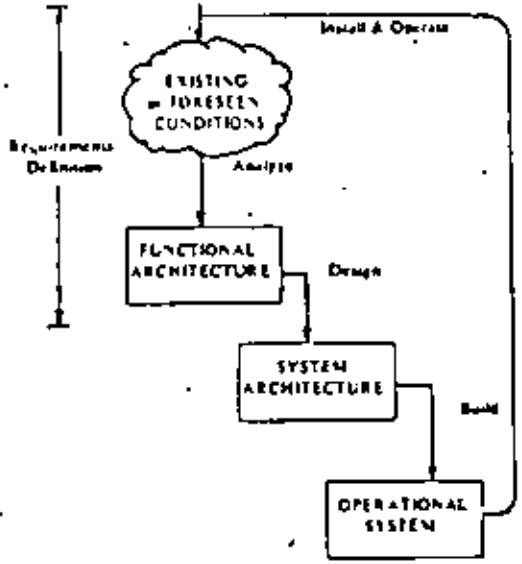


Figure 1. Simplified view of development cycle.

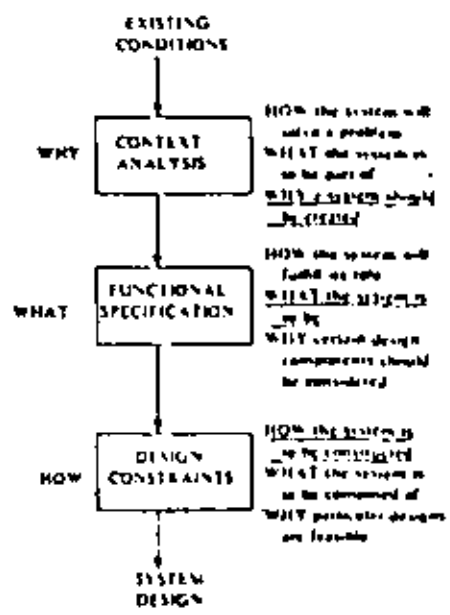


Figure 2. Each subject has a fundamental purpose.

Each of the subjects has a specific and limited role in justifying the required system. Collectively, they capture on paper, at the appropriate time, all relevant knowledge about the system problem in a complete, concise, comprehensive form. Taken together, these subjects define the whole need. Separately, they say *what* the system is to be part of, *what* the system is to be, and of *what* the system is to be composed (Fig. 2). The process known as "analysis" must apply to all three.

Descriptions of these subjects just frame the problem; they do not solve it. Details are postponed, and no binding implementation decisions are yet made. Even detailed budget and schedule commitments are put off, except those for the next phase, system design, because they depend on the results of that design. Requirements definition only (but completely) provides *boundary conditions* for the subsequent design and implementation stages. A problem well-stated is well on its way to a sound solution.

The problem revisited

The question still remains: why is requirements definition not a standard part of every system project, especially since not doing it well has such disastrously high costs [2]? Why is project start-up something one muddles through and why does it seem that requirements are never completely stated? What goes wrong?

The answer seems to be that just about everything goes wrong. Stated requirements are often excessive, incomplete, and inconsistent. Communication and documentation are roadblocks, too. Because they speak with different vocabularies, users and developers find it difficult to completely understand each other. Analysts are often drawn from the development organization, and are unable to document user requirements without simultaneously stating a design approach.

Good requirements are complete, consistent, testable, traceable, feasible, and flexible. By just stating necessary boundary conditions, they leave room for tradeoffs during system design. Thus, a good set of requirements documents can, in effect, serve as a user-developer contract. To attain these attributes, simply proposing a table of contents for requirements documentation is not enough. To do the job, one must emphasize the *means* of defining requirements, rather than prescribe the contents of documentation (which would, in any case, be impossible to do in a generic way).

Even in organizations which do stipulate some document or another, the process of defining requirements remains laborious and inconclusive. Lacking a complete definition of the job to be done, the effect of a contract is lacking, and the designers will make the missing assumptions and decisions because they must, in order to get the job done. Even when the value of requirements definition is recognized, it takes more than determination to have an effective approach. To define requirements, one must understand: 1) the *nature* of that which is to be described, 2) the form of the description; and 3) the *process* of analysis.

Many remedies to these problems have been proposed. Each project management, analysis, or specification scheme, whether "structured" or not, has its own adherents. Most do indeed offer some improvements, for any positive steps are better than none. But a significant impact has not been achieved, because each partial solution omits enough of the essential ingredients to be vulnerable.

The key to successful requirements definition lies in remembering that people define requirements. Thus, any useful discussion of requirements definition must combine: 1) a generic understanding of systems which is scientifically sound; 2) a notation and structure of documenting specific system knowledge in a rigorous, easy-to-read form; 3) a process for doing analysis which includes definition of people roles and interpersonal procedures; and 4) a way to technically manage the work, which enables allocation of requirements and postponement of design.

Academic approaches won't do. A pragmatic methodology must itself be: 1) technically feasible, i.e., consistent with the systems to be developed; 2) operationally feasible, i.e., people will use it to do the job well; and 3) economically feasible, i.e., noticeably improve the system development cycle.

This paper sketches such an approach, which has been successful in bringing order and direction to a wide range of system contexts. In even the most trying of circumstances, something can be done to address the need for requirements definition.

II. The process of requirements definition

The nature of systems

One fundamental weakness in current approaches to requirements definition is an inability to see clearly what the problem is, much less measure it, envision workable solutions, or apply any sort of assessment.

It is common practice to think of system architecture in terms of devices, languages, transmission links, and record formats. Overview charts of computer systems typically contain references to programs, files, terminals, and processors. At the appropriate time in system development, this is quite proper. But as an initial basis for system thinking, it is premature and it blocks from view precisely the key idea that is essential to successful requirements definition — the algorithmic nature of all systems. This important concept can best be envisioned by giving it a new name, the *functional architecture*, as distinct from the system architecture.

Systems consist of things that perform activities that interact with other things, each such interaction constituting a happening. A functional architecture is a rigorous layout of the activities performed by a system (temporarily disregarding who or what does them) and the things with which those activities interact. Given this, a design process will create a system architecture which implements, in good order, the functions of the functional architecture. Requirements definition is founded on showing what the functional architecture is (Fig. 3), also showing why it is what it is, and constraining how the system architecture is to realize it in more concrete form.

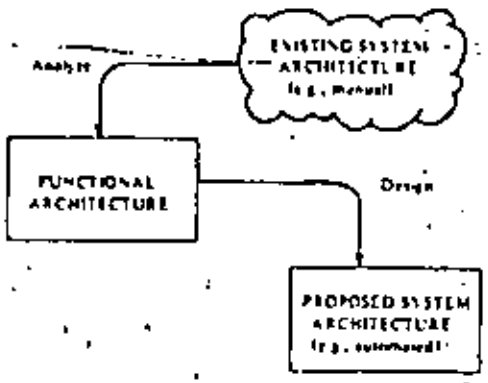


Figure 3. Functional architecture is extracted by analysis.

The concepts of functional architecture are universally applicable, to manual as well as automated systems, and are perfectly suited to the multiple needs of context analysis, functional specification, and design constraints found in requirements definition. Suppose, for example, that an operation, currently being performed manually is to be automated. The manual operation has a system architecture, composed of people, organizations, forms, procedures, and incentives, even though no computer is involved. It also has a functional architecture outlining the purposes for which the system exists. An automated system will implement the functional architecture with a different system architecture. In requirements definition, we must be able to extract the functional architecture (functional specification), and link it both to the boundary conditions for the manual operation (context analysis) and to the boundary conditions for the automated system (design constraints).

Functional architecture is founded on a generic universe of *things* and *happenings*:

objects	operations
data	activities
nouns	verbs
information	processing
substances	events
passive	active

Things and happenings are so intimately related that they can only exist together. A functional architecture always has a very strong structure making explicit these relationships. Functional architecture is, perhaps surprisingly, both abstract and precise.

Precision in functional architecture is best achieved by emphasizing only one primary structural relationship over and over — that of parts and wholes. Parts are related by interfaces to constitute wholes which, in turn, are parts of still larger wholes. It is always valid to express a functional architecture from the generic view that systems are made of components (parts and interfaces) and yet are themselves components.

So like all other architectures, the structure of functional architecture is both modular and hierarchic (even draftsmen and watchmakers use "top down" methods). Like other architectures, it can be seen from different viewpoints (even the construction trades use distinct structural, electrical, heating, and plumbing blueprints). And like other architectures, it may not necessarily be charted as a physical system would be (even a circuit diagram does not show the actual layout of components, but every important electrical characteristic is represented).

This universal way to view any system is the key to successful requirements definition. From our experience, people do not tend to do this naturally, and not in an organized fashion. In fact, we find that the most basic problem in requirements definition is the fact that most people do not even realize that such universality exists! When it is explained, the usual reaction is, "that is just common sense." But, as has been remarked for ages, common sense is an uncommon commodity. The need is to structure such concepts within a discipline which can be learned.

The form of documentation

To adequately define requirements, one must certainly realize that functional architecture exists, can be measured, and can be evaluated. But to describe it, one needs a communication medium corresponding to the blueprints and specifications that allow manufacturing to function smoothly. In fact, the form of documentation is the key to achieving communication. "Form" includes paging, paragraphing, use of graphics, document organization, and so forth. Because the distinction between form and content is so poorly understood, many adequate system descriptions are unreadable, simply because they are so hard to follow. When Marshall McLuhan said, "The medium is the message," he was apparently ignored by most system analysts.

Analysis of functional architecture (and design of system architecture) cannot be expressed both concisely and unambiguously in natural language. But by imbedding natural language in a blueprint-like graphic framework, all necessary relationships can be shown compactly and rigorously. Well-chosen graphic components permit representation of all aspects of the architecture in an artificial language which is natural for the specific function of communicating that architecture.

The universal nature of systems being both wholes and parts can be expressed by a graphic structure which is both modular and hierarchic (Fig. 4). Because parts are constrained to be wholes, interfaces must be shown explicitly. Interface notation must allow one to distinguish input from output from control (the concept of "control" will not be defended here). Most important, the notation itself must distinguish the things with which activities interact from the things which perform the activities. And because the graphics are a framework for the descriptive abilities of natural language, one must be able to name everything.

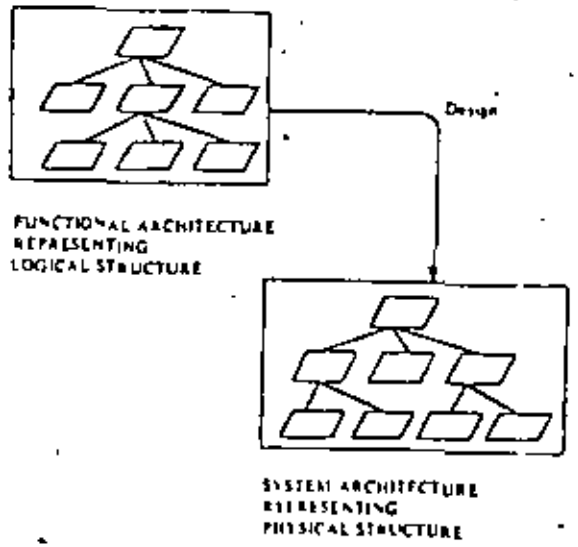


Figure 4. Physical structure is seldom identical to logical structure.

To achieve communication, the form of the diagram in which graphic symbols appear is also important. They must be bounded — to a single page or pair of facing pages — so that a reader can see at once everything which can be said about something. Each topic must be carefully delineated so a reader can grasp the whole message. A reader must be able to mentally walk through the architectural structure which is portrayed, just as blueprints enable a manufacturer to "see" the parts working together. And finally, everything must be indexed so that the whole set of diagrams will form a complete model of the architecture.

This appears to be a tall order, but when done elegantly, it yields documentation that is clear, complete, concise, consistent, and convincing. In addition, the hierarchic structure of the documentation can be exploited both to do and to manage the process of analysis. By reviewing the emergent documentation incrementally, for example, while requirements are being delineated, all interested parties can have a voice in directing the process. This is one of the features that results in the standardized, business-as-usual, "no surprises" approach found in manufacturing.

The analysis team

When thinking about why requirements are neither well-structured nor well-documented, one must not forget that any proposed methodology must be people-oriented. Technical matters matter very much, but it is the wishes, ideas, needs, concerns, and skills of people that determine the outcome. Technical aspects can only be addressed through the interaction of all people who have an interest in the system. One of the current difficulties in requirements definition, remember, is that system developers are often charged with documenting requirements. Their design background leads them (however well-intentioned they may be) to think of system architecture rather than functional architecture, and to define requirements in terms of solutions.

Consider a typical set of people who must actively participate in requirements definition. The *customer* is an organization with a need for a system. That customer authorizes a *commissioner* to acquire a system which will be operated by users. The commissioner, although perhaps technically oriented, probably knows less about system technology than the *developers* who will construct the system. These four parties may or may not be within one organization. For each administrative structure, there is a *management* group. Requirements definition must be understood by all these parties, answer the questions they have about the system, and serve as the basis for a development contract.

Each of these parties is a partisan whose conflicting, and often vague, desires must be amalgamated through requirements definition. There is a need at the center for trained, professional *analysts* who act as a catalyst to get the assorted information on paper and to structure from it adequate requirements documentation. The mental facility to comprehend abstraction, the ability to communicate it with personal tact, along with the ability to accept and deliver valid criticism, are all hallmarks of a professional analyst.

Analysts, many of whom may be only part-time, are not expected to supply expertise in all aspects of the problem area. As professionals, they are expected to seek out requirements from experts among the other parties concerned. To succeed, the task of analysis must be properly managed and coordi-

nated, and the requirements definition effort must embody multiple viewpoints. These viewpoints may be overlapping and, occasionally, contradictory.

Managing the analysis

Because many interests are involved, requirements definition must serve multiple purposes. Each subject — context analysis, functional specification, and design constraints — must be examined from at least three points of view: technical, operational, and economic. Technical assessment or feasibility concerns the architecture of a system. Operational assessment concerns the performance of that system in a working environment. Economic feasibility concerns the costs and impacts of system implementation and use. The point is simply that a wide variety of topics must be considered in requirements definition (Fig. 5). Without a plan for assembling the pieces into coherent requirements, it is easy for the analysis team to lose their sense of direction and oversteep their responsibility.

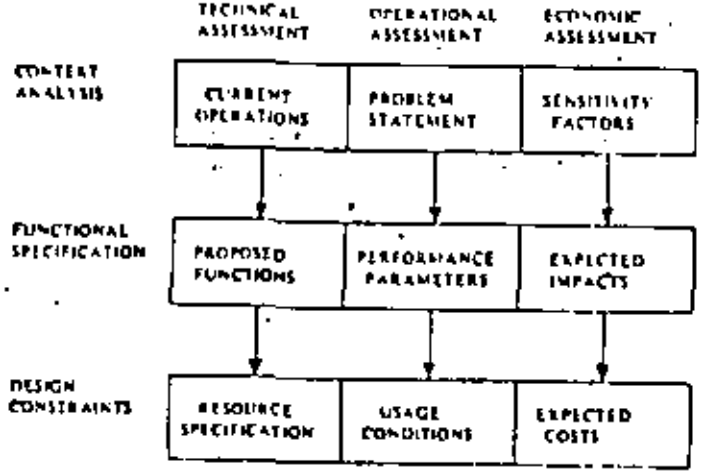


Figure 5. Multiple viewpoints of requirements definition.

Requirements definition, like all system development stages, should be an orderly progression. Each task is a logical successor to what has come before and is completed before proceeding to what comes after. Throughout, one must be able to answer the same three questions. 1) What are we doing? 2) Why are we doing it? 3) How do we proceed from here? Adequate management comes from asking these questions, iteratively, at every point in the development process. And if management is not to be a chameleon-like art, the same body of procedural knowledge should be applicable every time, although the subject at hand will differ.

It is precisely the lack of guidance about the process of analysis that makes requirements definition such a "hot item" today. People need to think about truly analyzing problems ("divide and conquer"), secure in the knowledge that postponed decisions will dovetail because the process they use enforces consistency. Analysis is an art of considering everything relevant at a given point, and nothing more. Adequate requirements will be complete, without over-specification, and will postpone certain decisions to later stages in the system development process without artifice. This is especially important in the development of those complex systems where requirements are imposed by higher level considerations (Fig. 6). Decisions must be *allocated* (i.e., postponed) because too many things interrelate in too many complicated ways for people to understand, all at once, what is and is not being said.

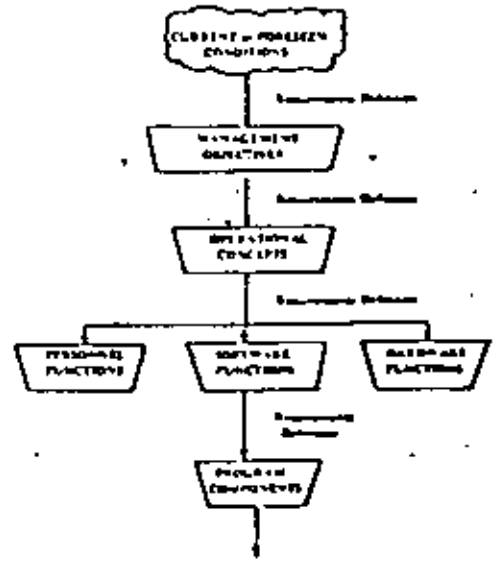


Figure 6. Analysis is repetitive in a complex environment.

Controlling a system development project is nearly impossible without reviews, walk-throughs, and configuration management. Such techniques become workable when the need for synthesis is recognized. Quite simply, system architectures and allocated requirements must be justifiable in light of previously stated requirements. One may choose, by plan or by default, not to enforce such traceability. However, validation and verification of subsequent project stages must not be precluded by ill-structured and unfathomable requirements.

Obviously, decisions on paper are the only ones that count. Knowing that an alternative was considered and rejected, and why, may often be as important as the final requirement. Full documentation becomes doubly necessary when the many parties involved are geographically separated and when staff turnover or expansion may occur before the project is completed.

The features just discussed at length — functional architecture, documentation, analysis teamwork, and the orderly process of analyzing and synthesizing multiple viewpoints — all must be integrated when prescribing a methodology for requirements definition.

III. Structured analysis

Outline of the approach

For several years, the senior author and his colleagues at SofTech have been developing, applying, and improving a general, but practical approach to handling complex system problems. The method is called Structured Analysis and Design Technique (SADT®) [3]. It has been used successfully on a wide range of problems by both SofTech and clients. This paper has presented some of the reasons why SADT works so well, when properly applied.

SADT evolved naturally from earlier work on the foundations of software engineering. It consists of both techniques for performing system analysis and design, and a process for applying these techniques in requirements definition and system development. Both features significantly increase the productivity and effectiveness of teams of people involved in a system project. Specifically, SADT provides methods for: 1) thinking in a structured way about large and complex problems; 2) working as a team with effective division and coordination of effort and roles; 3) communicating interview, analysis, and design results in clear, precise notation; 4) documenting current results and decisions in a way which provides a complete audit of history; 5) controlling accuracy, completeness and quality through frequent review and approval procedure; and 6) planning, managing, and assessing progress of the team effort. Two aspects of SADT deserve special mention: the graphic techniques and the definition of personnel roles.

Graphic techniques

The SADT graphic language provides a limited set of primitive constructs from which analysts and designers can compose orderly structures of any required size. The notation is quite simple — just boxes and arrows. *Boxes* represent parts of a whole in a precise manner. *Arrows* represent interfaces between parts. *Diagrams* represent wholes and are composed of boxes, arrows, natural language names, and certain other notations. The same graphics are applicable to both activities and data.

An SADT model is an organized sequence of diagrams, each with concise supporting text. A high-level overview diagram represents the whole subject. Each lower level diagram shows a limited amount of detail about a well-constrained topic. Further, each lower level diagram connects exactly into higher level portions of the model, thus preserving the logical relationship of each component to the total system (Fig. 7).

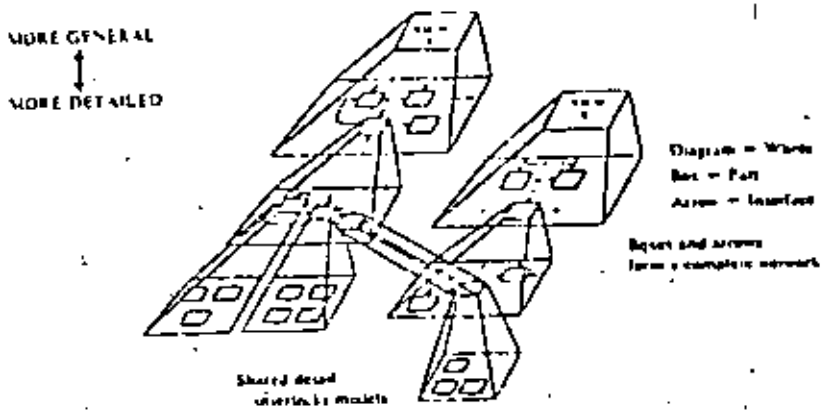


Figure 7. SADT provides practical, rigorous decomposition.

An SADT model is a graphic representation of the hierarchic structure of a system, decomposed with a firm purpose in mind. A model is structured so that it gradually exposes more and more detail. But its depth is bounded by the restriction of its vantage point and its content is bounded by its viewpoint. The priorities dictated by its purpose determine the layering of the top-down decomposition. Multiple models accommodate both multiple viewpoints and the various stages of system realization.

The arrow structure on an SADT diagram represents a constraint relationship among the boxes. It does not represent flow of control or sequence, as for example, on a flowchart for a computer program. Constraint arrows show necessary conditions imposed on a box.

Most arrows represent interfaces between boxes, whether in the same or different models. Some arrows represent noninterface interlocking between models. Together, these concepts achieve both overlapping of multiple viewpoints and the desirable attributes of good analysis and design projects (e.g., modularity, flexibility, and so forth [4]). The interface structure, particularly, passes through several levels of diagrams, creating a web that integrates all parts of the decomposition and shows the whole system's environmental interfaces with the topmost box.

Process interaction

Clearly, requirements definition requires cooperative teamwork from many people. This in turn demands a clear definition of the kinds of interactions which should occur between the personnel involved. SADT anticipates this need by establishing titles and functions of appropriate roles (Fig. 8). In a requirements definition effort, for example, the "authors" would be analysts trained and experienced in SADT.

Name	Function
Authors	Personnel who study requirements and commissioner analyze the new functions and represent them by models based on SADT diagrams
Commenters	Usually authors, who must review and comment in writing on the work of other authors
Readers	Personnel who read SADT diagrams for information but are not expected to make written comments
Experts	Persons from whom authors obtain specialized information about requirements and constraints by means of interviews
Technical Committee	A group of senior technical personnel assigned to review the analysis at every major level of decomposition. They either resolve technical issues or recommend a decision to the project management
Project Librarian	A person assigned the responsibility of maintaining a centralized file of all project documents, making errors, distributing readers lists, keeping records, etc.
Project Manager	The member of the project who has the final technical responsibility for carrying out the system analysis and design.
Monitor for Chief Analyst	A person fluent in SADT who assists and advises project personnel in the use and application of SADT.
Instructor	A person fluent in SADT, who trains Authors and Commenters using SADT for the first time

Figure 8. Personnel roles for SADT.

The SADT process, in which these roles interact, meets the needs of requirements definition for continuous and effective communication, for understandable and current documentation, and for regular and critical review. The process exploits the structure of an SADT model so that decisions can be seen in context and can be challenged while alternatives are still viable.

Throughout a project, draft versions of diagrams in evolving models are distributed to project members for review. Commenters make their suggestions in writing directly on copies of the diagrams. Written records of decisions and alternatives are retained as they unfold. As changes and corrections are made, all versions are entered in the project files. A project librarian provides filings, distribution, and record-keeping support, and, not so incidentally, also ensures configuration control.

This process documents all decisions and the reasons why decisions were made. When commenters and authors reach an understanding, the work is reviewed by a committee of senior technical and management personnel. During the process, incorrect or unacceptable results are usually spotted early, and oversights or errors are detected before they can cause major disruptions. Since everything is on record, future enhancement and system maintenance can reference previous analysis and design decisions.

When documentation is produced as the model evolves, the status of the project becomes highly visible. Management can study the requirements for the design in a "top-down" manner, beginning with an overview and continuing to any relevant level of detail. Although presentations to upper management usually follow standard summary and walk-through methods, even senior executives sometimes become direct readers, for the blueprint language SADT is easily learned.

Implementing the approach

How the ideas discussed in this paper are employed will vary according to organization needs and the kinds of systems under consideration. The methodology which has been described is not just a theory, however, and has been applied to a wide range of complex problems from real-time communications to process control to commercial EDP to organization planning. It is, in fact, a total systems methodology, and not merely a software technique. ITT Europe, for example, has used SADT since early 1974 for analysis and design of both hardware/software systems (telephonic and telegraphic switches) and non-software people-oriented problems (project management and customer engineering). Other users exhibit similar diversity, from manufacturing (AFCAM [5]) to military training (TRAIDEX [6]). Users report that it is a communications vehicle which focuses attention on well-defined topics, that it increases management control through visibility and standardization, that it creates a systematic work breakdown structure for project teams, and that it minimizes errors through disciplined flexibility.

There is no set pattern among different organizations for the contents of requirements documentation. In each case, the needs of the users, the commissioner, and the development organization must be accommodated. Government agencies tend to have fixed standards, while other organizations encourage flexibility. In a numerical control application, almost 40 models were generated in requirements definition (SINTEF, University of Trondheim, Norway). At least one supplier of large-scale computer-based systems mandates consideration of system, hardware, software, commercial, and administrative

constraints. Among all distinct viewpoints, the only common ground lies in the vantage points of context analysis, functional specification, and design constraints. Experience has shown that use of well-structured models together with a well-defined process of analysis, when properly carried out, does provide a strong foundation for actual system design [7].

Because local needs are diverse, implementation of the approach discussed in this paper cannot be accomplished solely by the publication of policy or standards: It is very much a "learn by doing" experience in which project personnel acquire ways of understanding the generic nature of systems. One must recognize that a common sense approach to system manufacturing is not now widely appreciated. A change in the ways that people think about systems and about the systems work that they themselves perform cannot be taught or disseminated in a short period of time.

Further developments

In manufacturing enterprises, blueprinting and specification methods evolved long before design support tools. In contrast to that analogy, a number of current efforts have produced systems for automating requirements information, independent of the definition and verification methodology provided by SADT. To date, SADT applications have been successfully carried out manually, but in large projects, where many analysts are involved and frequent changes do occur, the question is not whether or how to automate but simply what to automate.

Existing computer tools which wholly or partly apply to requirements [8] are characterized by a specification (or a design) database which, once input, may be manipulated. All such attempts, however, begin with user requirements recorded in a machine-readable form. Two impediments immediately become evident. The first is that requirements stated in prose texts cannot be translated in a straightforward manner to interface with an automated problem language. The second is that no computer tool will ever perform the process of requirements definition. Defining and verifying requirements is a task done by users and analysts [9].

Given the right kind of information, however, computer tools can provide capabilities to insure consistency, traceability, and allocation of requirements. A good example of this match to SADT is PSL/PSA, a system resulting from several years' effort in the ISDOS Project at the University of Michigan [10, 11]. The PSL database can represent almost every relationship which appears on SADT diagrams, and the input process from diagrams to database can be done by a project librarian. Not only does SADT enhance human communication (between user and analyst and between analyst and designer), but the diagrams become machine-readable in a very straightforward manner.

The PSA data analyzer and report generator is useful for summarizing database contents and can provide a means of controlling revisions. If the database has been derived by SADT, enhancements to existing PSA capabilities are possible to further exploit the structure of SADT models. For example, SADT diagrams are not flow diagrams, but the interface constraints are directed. These precedence relations, systematically pursued in SADT to specify quantities (volumes and rates) and sequences, permit any desired degree of simulation of a model (whether performed mentally or otherwise).

Computer aids are created as support tools. SADT provides a total context, within which certain automated procedures can play a complementary role. The result will be a complete, systematized approach which both suits the needs of the people involved and enables automation to be used in and extended beyond requirements definition. Such comprehensive methods will enable arbitrary systems work to attain the fulfillment that blueprint techniques deliver in traditional manufacturing.

IV. Conclusion

Requirements definition is an important source of costly oversights in present-day system innovation, but something can be done about it. None of the thoughts presented here are mere speculation. All have produced real achievements. The methods described have been successfully applied to a wide range of planning, analysis and design problems involving men, machines, software, hardware, databases, communications, procedures, and finances. The significance of the methods seems to be that a well-structured approach to documenting what someone thinks about a problem or a system can materially aid both that person's thinking and his ability to convey his understanding to others. Properly channeled, the mind of man is indeed formidable. But only by considering all aspects of the task ahead can teamwork be more productive and management be more effective. Communication with nontechnical readers, an understanding of the nature and structure of systems [12], and indeed a thorough knowledge of the process of analysis itself are the essential ingredients of successful requirements definition.

Appendix

Asking the right questions

The basic difficulty in requirements definition is not one of seeing the woods instead of the trees. The real need is for a methodology which, in any given circumstance, will enable an analyst to distinguish one from the other. It is always more important to ask the right questions than it is to assert possibly wrong answers. It is said that when a famous rabbi was asked, "Rabbi, why does a rabbi always answer a question with a question?" he replied, "Is there a better way?" This is the famous dialectic method used by Socrates to lead his

students to understanding. Answering questions with questions leaves options open, and has the nature of breaking big questions into a top-down structure of smaller questions, easier and easier to address. The answers, when they are ultimately supplied, are each less critical and more tractable, and their relations with other small answers are well-structured.

This is the focus of requirement definition. The appropriate questions — *why, what, how* — applied systematically, will distinguish that which must be considered from that which must be postponed. A sequence of such questions, on a global, system-wide scale, will break the complexity of various aspects of the system into simpler, related questions which can be analyzed, developed, and documented. The context analysis, functional specification, and design constraints — subjects which are part of requirements definition — are merely parts of an overlapping chain of responses to the appropriate why, what, how questions (Fig. 9). In different circumstances, the subjects may differ, but the chaining of questions will remain the same. *Why* some feature is needed molds *what* it has to be, which in turn molds *how* it is to be achieved.

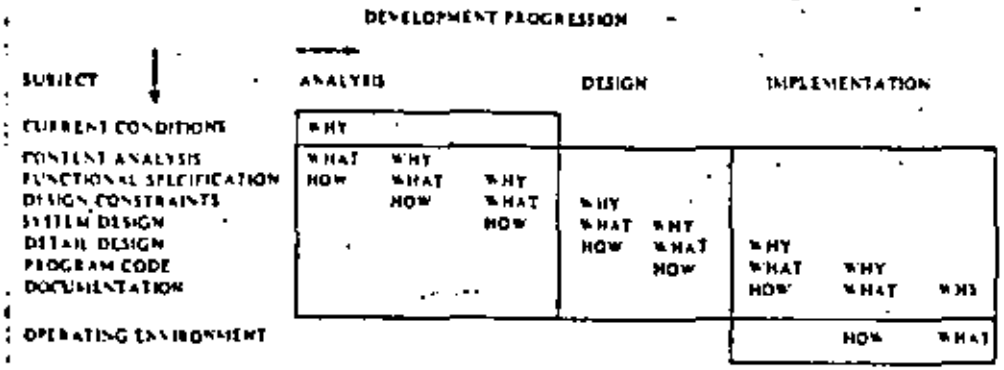


Figure 9. System development is a chain of overlapping questions, documented at each step.

These questions form an overlapping repetition of a common pattern. Each time, the various aspects of the system are partitioned, and the understanding which is developed must be documented in a form consistent with the pattern. It is not sufficient merely to break big problems into little problems by shattering them. "Decompose" is the inverse of "compose"; at every step, the parts being considered must reassemble to make the whole context within which one is working.

The English word "cleave" captures the concept exactly. It is one of those rare words that has antithetical meanings. It means both to separate and to cling to! Thus, in the orderly process of top-down decomposition which describes the desired system, multiple views must intersect in order to supply the whole context for the next stage of system development (Fig. 10).

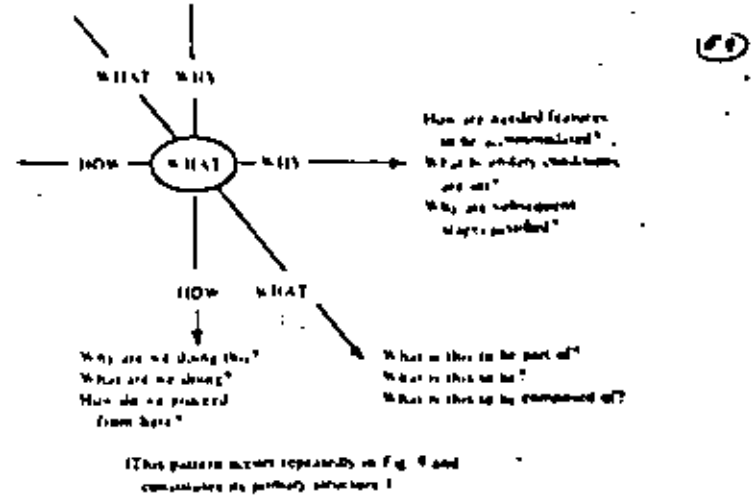


Figure 10. Right questions occur within a context and form a context as well.

And finding the right answers

Probably the most important aspect of this paper is its emphasis on a common approach to all phases of system development, starting with requirements definition. Knowing how postponed decisions will be handled in later phases (by the same methods) allows their essence to be established by boundary conditions, while details are left open (Fig. 11). The knowledge that all requirements must ultimately be implemented somehow (again by the same methods) allows their completeness and consistency to be verified. Finally, an orderly sequence of questions enforces gradual exposition of detail. All information is presented in well-structured documentation which allows first-hand participation by users and commissioners in requirements definition.

The way to achieve such coherence is to seek the right kind of answers to every set of why, what, how questions. By this is meant to establish the viewpoint, vantage point, and purpose of the immediate task before writing any document or conducting any analysis.

The initial *purpose* of the system development effort is established by context analysis. Proper cleaving of subjects and descriptions then takes two different forms, the exact natures of which are governed by the overall project purpose. One cleaving creates partial but overlapping delineations according to *viewpoint* — viewpoint makes clear what aspects are considered relevant to achieving some component of the overall purpose. The other cleaving creates rigorous functional architectures according to *vantage point* — vantage point is a level of abstraction that relates the viewpoints and component purposes which together describe a given whole subject (Fig. 12).

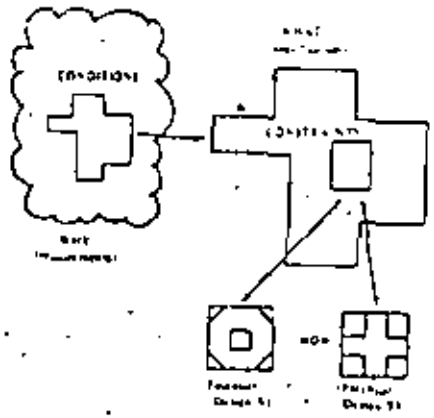


Figure 11. Postponed decisions occur within a prior framework.

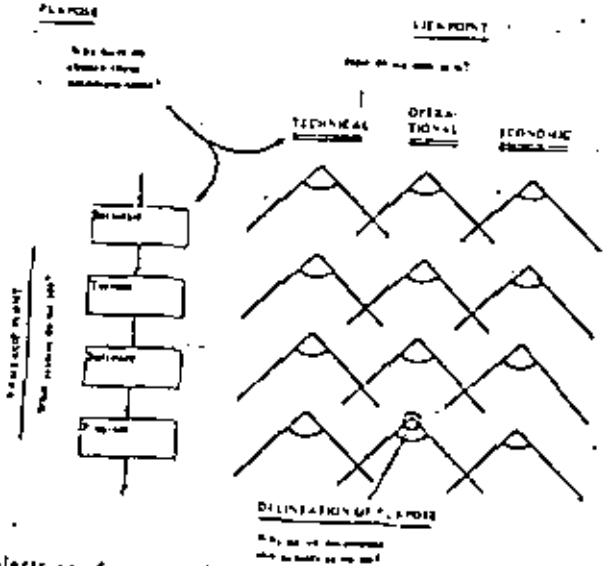


Figure 12. Subjects are decomposed according to viewpoint, vantage point, and purpose.

Depending on the system, any number of viewpoints may be important in requirements definition and may continue to be relevant as the vantage point shifts to designing, building, and finally using the system. Any single viewpoint always highlights some aspects of a subject, while other aspects will be lost from view. For example, the same system may be considered from separate viewpoints which emphasize physical characteristics, functional characteristics, operation, management, performance, maintenance, construction cost, and so forth.

Picking a vantage point always abstracts a functional architecture, while muting implementation-defined aspects of the system architecture. The placement of a vantage point within a viewpoint establishes an all-important *bounded context* — a subset of purpose which then governs the decomposition and exposition of a particular subject regarding the system.

Can give the right results

Requirements definition, beginning with context analysis, can occur whenever there is a need to define, redefine, or further delineate purpose. Context analysis treats the most important, highest level questions first, setting the conditions for further questioning each part in turn. Each subsequent question is asked within the bounded context of a prior response. Strict discipline ensures that each aspect of any question is covered by exactly one further question. Getting started is difficult, but having done so successfully, one is introduced to a "top-down" hierarchy of leading questions, properly sequenced, completely decomposed, and successively answered.

Whether explicitly stated or not, vantage points, viewpoints, and purposes guide the activities of *any* analysis team which approaches a requirements definition task. With the global understanding offered by the above discussion, analysts should realize that a place can be found for every item of information gathered. Structuring this mass of information still must be done, on paper, in a way that ensures completeness and correctness, without overspecification and confusion. That is the role of SADT.

Acknowledgment

The authors would like to thank J.W. Brackett and J.B. Goodenough of SofTech, Inc., Waltham, MA, who made several helpful suggestions incorporated into the presentation of these ideas. Many people at SofTech have, of course, contributed to the development and use of SADT.

1. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5 (May 1973), pp. 48-59.

2. P. Hirsch, "GAO Hits Winnmix Hard: FY'72 Funding Prospects Fading Fast," *Datamation*, Vol. 17, No. 7 (March 1, 1971), p. 41.

3. *An Introduction to SADT™*, SofTech, Inc., Report No. 9022-78 (Waltham, Mass.: February 1976).

4. D.T. Ross, J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process, Principles, and Goals," *Computer*, Vol. 8, No. 5 (May 1975), pp. 17-27.

5. Air Force Materials Laboratory, *Air Force Computer-Aided Manufacturing (AFCAM) Master Plan*, Vol. II, App. A, and Vol. III, AFSC, Wright Patterson Air Force Base, Report No. AFML-TR-74-104 (Ohio: July 1974). Available from DDC as AD 922-041L and 922-171L.

6. *TRAIDEX Needs and Implementation Study*, SofTech, Inc., Final Report (Waltham, Mass.: May 1976). Available as No. ED-129244 from ERIC Printing House on Information Resources (Stanford, Calif.).

7. B.W. Boehm, "Software Design and Structure," *Practical Strategies for Developing Large Software Systems*, ed. E. Horowitz (Reading, Mass.: Addison-Wesley, 1975), pp. 115-22.

8. R.V. Head, "Automated System Analysis," *Datamation*, Vol. 17, No. 16 (Aug. 15, 1971), pp. 22-24.

9. J.T. Rigo, "How to Prepare Functional Specifications," *Datamation*, Vol. 20, No. 5 (May 1974), pp. 78-80.

10. D. Teichroew and H. Sayani, "Automation of System Building," *Datamation*, Vol. 17, No. 16 (Aug. 15, 1971), pp. 25-30.

11. D. Teichroew and E.A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 41-48.

12. F.M. Haney, "What It Takes to Make MAC, MIS, and ABM Fly," *Datamation*, Vol. 20, No. 6 (June 1974), pp. 168-69.

INTRODUCTION 63

I vividly remember a *Datamation* article, written by Daniel Teichroew in 1967, predicting that within ten years programmers would be obsolete. All we had to do, he said, was make it possible for users to state *precisely* what they wanted a computer system to do for them; at that point, it should be possible to mechanically generate the code. Having been in the computer field for only a few years, I was profoundly worried. Maybe it was time to abandon data processing and become a farmer?

Those early ideas of Professor Teichroew perhaps were ahead of their time — after all, there still were a sizable number of programmers in existence in 1977! — but some of his predictions are beginning to take concrete form today. The following paper, written by Teichroew and Hershey and originally published in the January 1977 *IEEE Transactions on Software Engineering*, is the best single source of information on a system for automated analysis, known as "ISDOS" or "PSL/PSA."

The Teichroew/Hershey paper deals specifically with structured analysis, and, as such, is radically different from the earlier papers on programming and design. Throughout the 1960s, it was fashionable to blame all of our EDP problems on *programming*; structured programming and the related disciplines were deemed the ideal solution. Then, by the mid-1970s, emphasis shifted toward *design*, as people began to realize that brilliant code would not save a mediocre design. But now our emphasis has shifted even further: Without an adequate statement of the user's requirements, the best design and the best code in the world won't do the job. Indeed, without benefit of proper requirements definition, structured design and structured programming may simply help us arrive at a systems disaster faster.

So, what can be done to improve the requirements definition process? There are, of course, the methods proposed by Ross and Schoman in the previous paper, as well as those set forth by DeMarco in the paper that appears after this one. In this paper, Teichroew and Hershey concentrate on the *documentation* associated with re-

quirements definition, and on the difficulty of producing and managing manually generated documentation.

The problems that Teichroew and Hershey address certainly are familiar ones, although many systems analysis probably have come to the sad conclusion that things always were meant to be like this. They observe, for example, that much of the documentation associated with an EDP system is not formally recorded until the *end* of the project, by which time, as DeMarco points out, the final document is "of historical significance only" (Paper 24). The documentation generated is notoriously ambiguous, verbose, and redundant, and, worst of all, it is *manually* produced, *manually* examined for possible errors and inconsistencies, and *manually* updated and revised as user requirements change.

So, the answer to all of this, in Teichroew and Hershey's opinion, is a computer program that allows the systems analyst to input the user requirements in a language that can be regarded as a subset of English; those requirements then can be changed, using facilities similar to those on any modern text-editing system, throughout the analysis phase of the project — *and throughout the entire system life cycle!* Perhaps most important, the requirements definition can be subjected to automated analysis to determine such things as contradictory definitions of data elements, missing definitions, and data elements that are generated but never used.

The concept for PSL was documented by Teichroew more than ten years ago, but it is in this 1977 paper that we begin to get some idea of the impact on the real world. Automated analysis slowly is becoming an option, with a number of large, prestigious organizations using PSL/PSA.

One would expect continued growth; ironically, though, a reported problem is *machine inefficiency!* PSL/PSA apparently consumes significant amounts of CPU time and other computer resources, although Teichroew and Hershey maintain that such tangible, visible costs probably are smaller than the intangible, hidden costs of time wasted during "manual" analysis. It appears that PSL/PSA is most successfully used in organizations that already have vast computer installations and several hundred (or thousand) EDP people. A significant drawback to widespread adoption of PSL/PSA is the fact that it is written in FORTRAN! Admittedly this has helped make it portable — but FORTRAN?

PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems

1. Introduction

Organizations now depend on computer-based information processing systems for many of the tasks involving data (recording, storing, retrieving, processing, etc.). Such systems are man-made, the process consists of a number of activities: perceiving a need for a system, determining what it should do for the organization, designing it, constructing and assembling the components, and finally testing the system prior to installing it. The process requires a great deal of effort, usually over a considerable period of time.

Throughout the life of a system it exists in several different "forms." Initially, the system exists as a concept or a proposal at a very high level of abstraction. At the point where it becomes operational it exists as a collection of rules and executable object programs in a particular computing environment. This environment consists of hardware and hard software such as the operating system, plus other components such as procedures which are carried out manually. In between the system exists in various intermediary forms.

The process by which the initial concept evolves into an operational system consists of a number of activities each of which makes the concept more concrete. Each activity takes the results of some of the previous activities and produces new results so that the progression

eventually results in an operational system. Most of the activities are data processing activities, in that they use data and information to produce other data and information. Each activity can be regarded as receiving specifications or requirements from preceding activities and producing data which are regarded as specifications or requirements by one or more succeeding activities.

Since many individuals may be involved in the system development process over considerable periods of time and these or other individuals have to maintain the system once it is operating, it is necessary to record descriptions of the system as it evolves. This is usually referred to as "documentation."

In practice, the emphasis in documentation is on describing the system in the final form so that it can be maintained. Ideally, however, each activity should be documented so that the results it produces become the specification for succeeding activities. This does not happen in practice because the communications from one activity to succeeding activities is accomplished either by having the same person carrying out the activities, by oral communication among individuals in a project, or by notes which are discarded after their initial use.

This results in projects which proceed without any real possibility for management review and control. The systems are not ready when promised, do not perform the function the users expected, and cost more than budgeted.

Most organizations, therefore, mandate that the system development process be divided into phases and that certain documentation be produced by the end of each phase so that progress can be monitored and corrections made when necessary. These attempts, however, leave much to be desired and most organizations are attempting to improve the methods by which they manage their system development [20, 6].

This paper is concerned with one approach to improving systems development. The approach is based on three premises. The first is that more effort and attention should be devoted to the front end of the process where a proposed system is being described from the user's point of view [2, 14, 3]. The second premise is that the computer should be used in the development process since systems development involves large amounts of information processing. The third premise is that a computer-aided approach to systems development must start with "documentation."

This paper describes a computer-aided technique for documentation which consists of the following:

1) The results of each of the activities in the system development process are recorded in computer processible form as they are produced.
2) A computerized data base is used to maintain all the basic data about the system.
3) The computer is used to produce hard copy documentation when required.

The part of the technique which is now operational is known as PSL/PSA. Section II is devoted to a brief description of system development as a framework in which to compare manual and computer-aided documentation methods. The Problem Statement Language (PSL) is described in Section III. The reports which can be produced by the Problem Statement Analyzer (PSA) are described in Section IV. The status of the system, results of experience to date, and planned developments are outlined in Section V.

II. Logical systems design

The computer-aided documentation system described in Sections III and IV of this paper is designed to play an integral role during the initial stages in the system development process. A generalized model of the whole system development process is given in Section II-A. The final result of the initial stage is a document which here will be called the System Definition Report. The desired contents of this document are discussed in Section II-B. The activities required to produce this document manually are described in Section II-C and the changes possible through the use of computer-aided methods are outlined in Section II-D.

A. A model of the system development process

The basic steps in the life cycle of information systems (initiation, analysis, design, construction, test, installation, operation, and termination) appeared in the earliest applications of computers to organizational problems (see for example, [17, 1, 4, 7]). The need for more formal and comprehensive procedures for carrying out the life cycle was recognized; early examples are the IBM SOP publications [5], the Philips ARDI method [8], and the SDC method [23]. In the last few years, a large number of books and papers on this subject have been published [11, 19].

Examination of these and many other publications indicate that there is no general agreement on what phases the development process should be divided into, what documentation should be produced at each phase, what it should contain, or what form it should be presented in. Each organization develops its own methods and standards.

In this section a generalized system development process will be described as it might be conducted in an organization which has a Systems Department responsible for developing, operating, and maintaining computer based information processing systems. The System Department belongs to some higher unit in the organization and itself has some subunits, each with certain func-

tions (see for example, [24]). The System Department has a system development standard procedure which includes a project management system and documentation standards.

A request for a new system is initiated by some unit in the organization or the system may be proposed by the System Department. An initial document is prepared which contains information about why a new system is needed and outlines its major functions. This document is reviewed and, if approved, a senior analyst is assigned to prepare a more detailed document. The analyst collects data by interviewing users and studying the present system. He then produces a report describing his proposed system and showing how it will satisfy the requirements. The report will also contain the implementation plan, benefit/cost analysis, and his recommendations. The report is reviewed by the various organizational units involved. If it passes this review it is then included with other requests for the resources of the System Department and given a priority. Up to this point the investment in the proposed system is relatively small.

At some point a project team is formed, a project leader and team members are assigned, and given authority to proceed with the development of the system. A steering group may also be formed. The project is assigned a schedule in accordance with the project management system and given a budget. The schedule will include one or more target dates. The final target date will be the date the system (or its first part if it is being done in parts) is to be operational. There may also be additional target dates such as beginning of system test, beginning of programming, etc.

B. Logical system design documentation

In this paper, it is assumed that the system development procedure requires that the proposed system be reviewed before a major investment is made in system construction. There will therefore be another target date at which the "logical" design of the proposed system is reviewed. On the basis of this review the decision may be to proceed with the physical design and construction, to revise the proposed system, or to terminate the project.

The review is usually based on a document prepared by the project team. Sometimes it may consist of more than one separate document; for example, in the systems development methodology used by the U.S. Department of Defense [21] for non-weapons systems, development of the life cycle is divided into phases. Two documents are produced at the end of the Definition subphase of the Development phase: a Functional Description, and a Data Requirements Document.

Examination of these and many documentation requirements show that a Systems Definition Report contains five major types of information:

1) a description of the organization and where the proposed system will fit, showing how the proposed system will improve the functioning of the organization or otherwise meet the needs which lead to the project;

2) a description of the operation of the proposed system in sufficient detail to allow the users to verify that it will in fact accomplish its objectives, and to serve as the specification for the design and construction of the proposed system if the project continuation is authorized;

3) a description of its proposed system implementation in sufficient detail to estimate the time and cost required;

4) the implementation plan in sufficient detail to estimate the cost of the proposed system and the time it will be available;

5) a benefit/cost analysis and recommendations.

In addition, the report usually also contains other miscellaneous information such as glossaries, etc.

C. Current logical system design process

During the initial stages of the project the efforts of the team are directed towards producing the Systems Definition Report. Since the major item this report contains is the description of the proposed system from the user or logical point of view, the activities required to produce the report are called the logical system design process. The project team will start with the information already available and then perform a set of activities. These may be grouped into five major categories.

1) *Data collection.* Information about the information flow in the present system, user desires for new information, potential new system organization, etc., is collected and recorded.

2) *Analysis.* The data that have been collected are summarized and analyzed. Errors, omissions, and ambiguities are identified and corrected. Redundancies are identified. The results are prepared for review by appropriate groups.

3) *Logical design.* Functions to be performed by the system are selected. Alternatives for a new system or modification of the present system are developed and examined. The "new" system is described.

4) *Evaluation.* The benefits and costs of the proposed system are determined to a suitable level of accuracy. The operational and functional feasibility of the system are examined and evaluated.

5) *Improvements.* Usually as a result of the evaluation a number of deficiencies in the proposed system will be discovered. Alternatives for improvement are identified and evaluated until further possible improvements are not judged to be worth additional effort. If major changes are made, the evaluation step may be repeated, further data collection and analysis may also be necessary.

In practice the type of activities outlined above may not be clearly distinguished and may be carried out in parallel or iteratively with increasing level of detail. Throughout the process, however it is carried out, results are recorded and documented.

It is widely accepted that documentation is a weak link in system development in general and in logical system design in particular. The representation in the documentation that is produced with present manual methods is limited to:

1) text in a natural language;
2) lists, tables, arrays, cross references;
3) graphical representation, figures, flowcharts.

Analysis of two reports showed the following number of pages for each type of information.

Form	Report A	Report B
text	90	117
lists and tables	207	165
charts and figures	78	54
total	335	336

The systems being documented are very complex and these methods of representation are not capable of adequately describing all the necessary aspects of a system for all those who must, or should, use the documentation. Consequently, documentation is

1) ambiguous: natural languages are not precise enough to describe systems and different readers may interpret a sentence in different ways;
2) inconsistent: since systems are large the documentation is large and it is very difficult to ensure that the documentation is consistent;
3) incomplete: there is usually not a sufficient amount of time to devote to documentation and with a large complex system it is difficult to determine what information is missing.

The deficiencies of manual documentation are compounded by the fact that systems are continually changing and it is very difficult to keep the documentation up-to-date.

Recently there have been attempts to improve manual documentation by developing more formal methodologies [16, 12, 13, 22, 15, 25]. These methods, even though they are designed to be used manually, have a formal language or representation scheme that is designed to alleviate the difficulties listed above. To make the documentation more useful for human beings, many of these methods use a graphical language.

D. Computer-aided logical system design process

In computer-aided logical system design the objective, as in the manual process, is to produce the System Definition Report and the process followed is essentially similar to that described above. The computer-aided design system has the following capabilities:

1) capability to describe information systems, whether manual or computerized, whether existing or proposed, regardless of application area;
2) ability to record such description in a computerized data base;
3) ability to incrementally add to, modify, or delete from the description in the data base;
4) ability to produce "hard copy" documentation for use by the analyst or the other users.

The capability to describe systems in computer processible form results from the use of the system description language called PSL. The ability to record such description in a data base, incrementally modify it, and on demand perform analysis and produce reports comes from the software package called the Problem Statement Analyzer (PSA). The Analyzer is controlled by a Command Language which is described in detail in [9] (Fig. 1).

The Problem Statement Language is outlined in Section III and described in detail in [10]. The use of PSL/PSA in computer-aided logical system design is described in detail in [18].

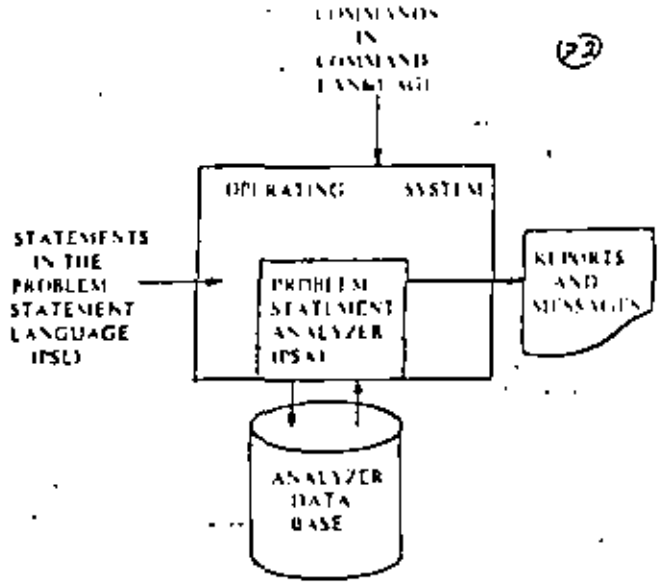


Figure 1. The problem statement analyzer.

The use of PSL/PSA does not depend on any particular structure of the system development process or any standards on the format and content of hard copy documentation. It is therefore fully compatible with current procedures in most organizations that are developing and maintaining systems. Using this system, the data collected or developed during all five of the activities are recorded in machine-readable form and entered into the computer as it is collected. A data base is built during the process. These data can be analyzed by computer programs and intermediate documentation prepared on request. The Systems Definition Report then includes a large amount of material produced automatically from the data base.

The activities in logical system design are modified when PSL/PSA is used as follows:

1) Data collection: since most of the data must be obtained through personal contact, interviews will still be required. The data collected are recorded in machine-readable form. The intermediate outputs of PSA also provide convenient checklists for deciding what additional information is needed and for recording it for input.

2) Analysis: a number of different kinds of analysis can be performed on demand by PSA, and therefore need no longer be done manually.

3) Design: design is essentially a creative process and cannot be automated. However, PSA can make more data available to the designer and allow him to manipulate it more extensively. The results of his decisions are also entered into the data base.

4) Evaluation: PSA provides some rudimentary facilities for computing volume or work measures from the data in the problem statement.

5) Improvements: identification of areas for possible improvements is also a creative task; however, PSA output, particularly from the evaluation phase, may be useful to the analyst.

The System Definition Report will contain the same material as that described since the documentation must serve the same purpose. Furthermore, the same general format and representation is desirable.

1) Narrative information is necessary for human readability. This is stored as part of the data but is not analyzed by the computer program. However, the fact that it is displayed next to, or in conjunction with, the final description improves the ability of the analyst to detect discrepancies and inconsistencies.

2) Lists, tables, arrays, matrices. These representations are prepared from the data base. They are up-to-date and can be more easily rearranged in any desired order.

3) Diagrams and charts. The information from the data base can be represented in various graphical forms to display the relationships between objects.

III. PSL, a problem statement language

PSL is a language for describing systems. Since it is intended to be used to describe "proposed" systems it was called a Problem Statement Language because the description of a proposed system can be considered a "problem" to be solved by the system designers and implementors.

PSL is intended to be used in situations in which analysts now describe systems. The descriptions of systems produced using PSL are used for the same purpose as that produced manually. PSL may be used both in batch and interactive environments, and therefore only "basic" information about the system need to be stated in PSL. All "derived" information can be produced in hard copy form as required.

The model on which PSL is based is described in Section III-A. A general description of the system and semantics of PSL is then given in Section III-B to illustrate the broad scope of system aspects that can be described using PSL. The detailed syntax of PSL in given in [10].

A. Model of information systems

The Problem Statement Language is based first on a model of a general system, and secondly on the specialization of the model to a particular class of systems, namely information systems.

The model of a general system is relatively simple. It merely states that a system consists of things which are called OBJECTS. These objects may have PROPERTIES and each of these PROPERTIES may have PROPERTY VALUES. The objects may be connected or interrelated in various ways. These connections are called RELATIONSHIPS.

The general model is specialized for an information system by allowing the use of only a limited number of predefined objects, properties, and relationships.

B. *An overview of the problem statement language syntax and semantics*

The objective of PSL is to be able to express in syntactically analyzable form as much of the information which commonly appears in System Definition Reports as possible.

System Descriptions may be divided into eight major aspects:

1) System Input/Output Flow,
2) System Structure,
3) Data Structure,
4) Data Derivation,
5) System Size and Volume,
6) System Dynamics,
7) System Properties,
8) Project Management.

PSL contains a number of types of objects and relationships which permit these different aspects to be described.

The *System Input/Output Flow* aspect of the system deals with the interaction between the target system and its environment.

System Structure is concerned with the hierarchies among objects in a system. Structures may also be introduced to facilitate a particular design approach such as "top down." All information may initially be grouped together and called by one name at the highest level, and then successively subdivided. System structures can represent high-level hierarchies which may not actually exist in the system, as well as those that do.

The *Data Structure* aspect of system description includes all the relationships which exist among data used and/or manipulated by the system as seen by the "users" of the system.

The *Data Derivation* aspect of the system description specifies which data objects are involved in particular PROCESSES in the system. It is concerned with what information is used, updated, and/or derived, how this is done, and by which processes.

Data Derivation relationships are internal in the system, while System Input/Output Flow relationships describe the system boundaries. As with other PSL facilities System Input/Output Flow need not be used. A system can be considered as having no boundary.

The *System Size and Volume* aspect is concerned with the size of the system and those factors which influence the volume of processing which will be required.

The *System Dynamics* aspect of system description presents the manner in which the target system "behaves" over time.

All objects (of a particular type) used to describe the target system have characteristics which distinguish them from other objects of the same type. Therefore, the PROPERTIES of particular objects in the system must be described. The PROPERTIES themselves are objects and given unique names.

The *Project Management* aspect requires that, in addition to the description of the target system being designed, documentation of the project designing (or documenting) the target system be given. This involves identification of people involved and their responsibilities, schedules, etc.

IV. Reports

As information about a particular system is obtained, it is expressed in PSL and entered into a data base using the Problem Statement Analyzer. At any time standard outputs or reports may be produced on request. The various reports can be classified on the basis of the purposes which they serve.

1) *Data Base Modification Reports:* These constitute a record of changes that have been made, together with diagnostics and warnings. They constitute a record of changes for error correction and recovery.

2) *Reference Reports:* These present the information in the data base in various formats. For example, the Name List Report presents all the objects in the data base with their type and date of last change. The Formatted Problem Statement Report shows all properties and relationships for a particular object

(Fig. 2). The Dictionary Report gives only data dictionary type inform

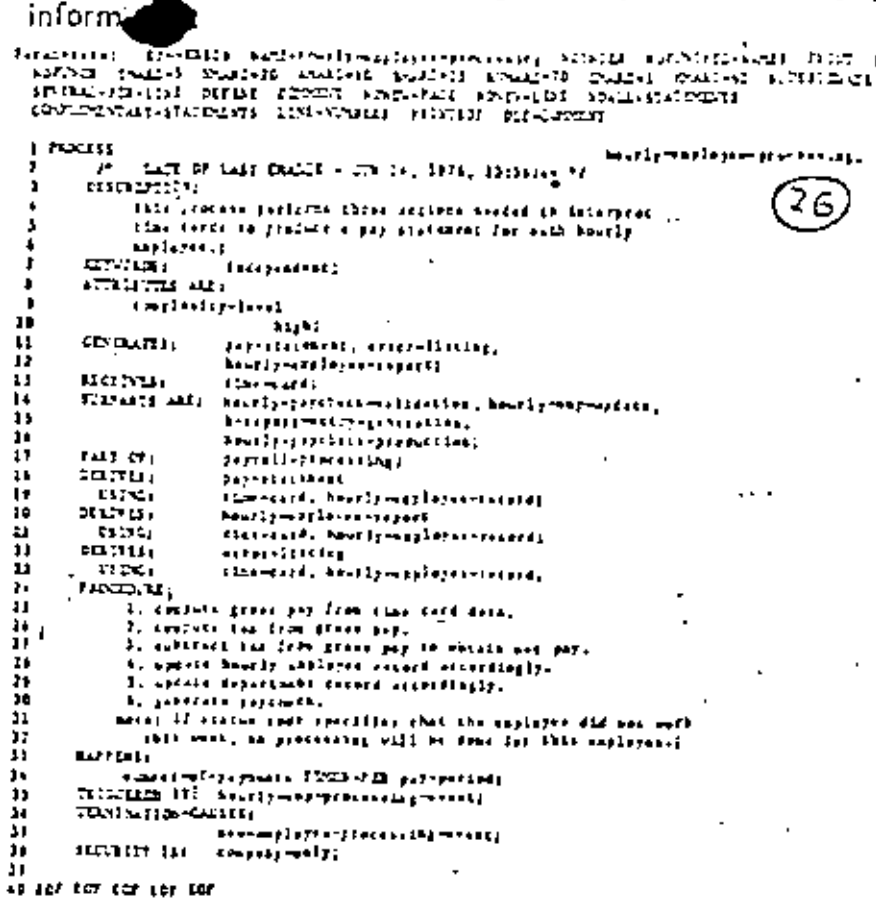


Figure 2. Example of a FORMATTED PROBLEM STATEMENT for one PROCESS.

Summary Reports: These present collections of information in summary from, or gathered from several different relationships. For example, the Data Base Summary Report provides project management information by showing the totals of various types of objects and how much has been said about them. The Structure Report shows complete or partial hierarchies. The Extended Picture Report shows the data flows in a graphical form.

Analysis Reports: These provide various types of analysis of the information in the data base. For example, the Contents Comparison Report analyzes similarity of Inputs and Outputs. The Data Process Interaction Report (Fig. 3) can be used to detect gaps in the information flow, or unused data objects. The Process Chain Report shows the dynamic behavior of the system (Fig. 4).

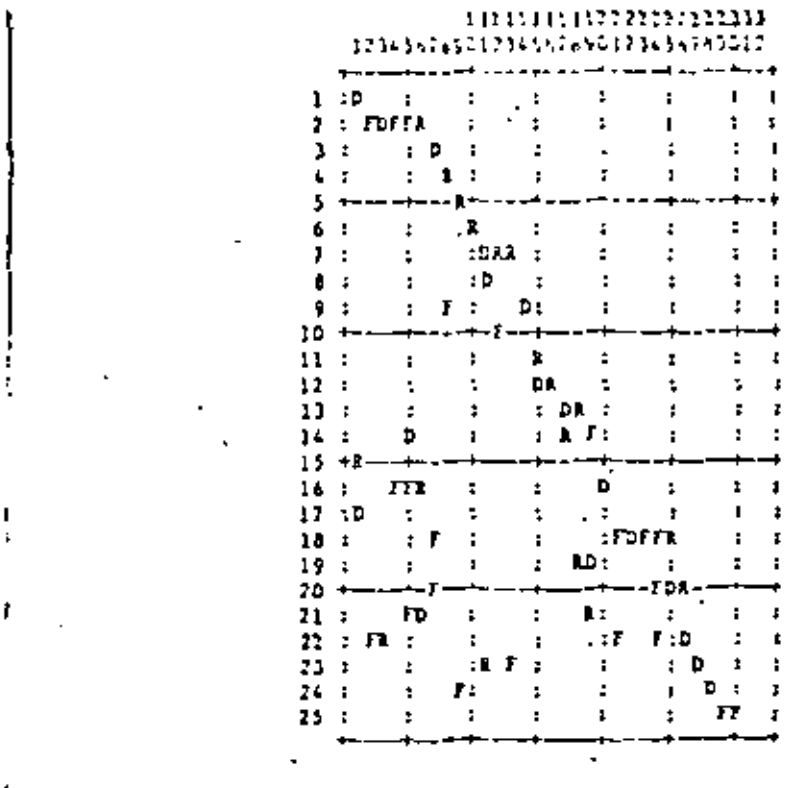


Figure 3. Example of part of a Data Process Interaction Report.

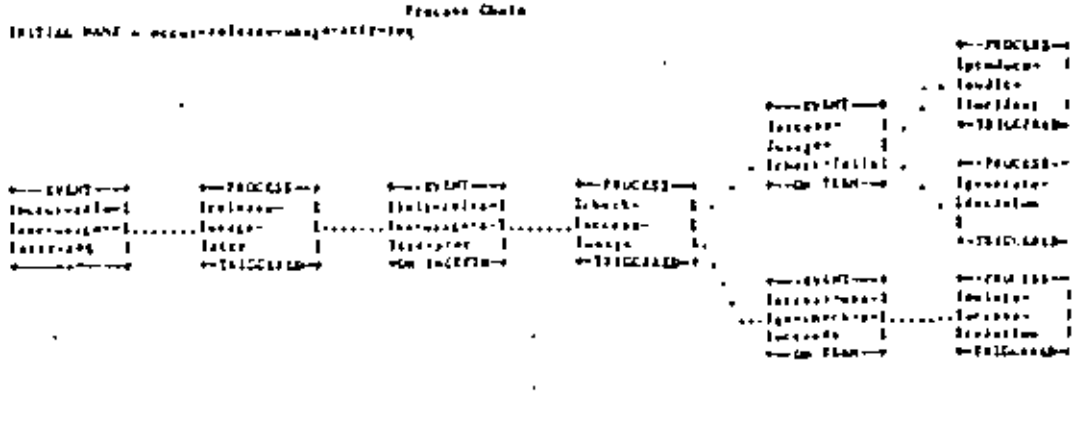


Figure 4. Example of a Process Chain Report.

After the requirements have been completed, the final documentation required by the organization can be produced semiautomatically to a prescribed format, e.g., the format required for the Functional Description and Data Requirements in [21].

V. Concluding remarks

The current status of PSL/PSA is described briefly in Section V-A. The benefits that should accrue to users of PSL/PSA are discussed in Section V-B. The information on benefits actually obtained by users is given in Section V-C. Planned extensions are outlined in Section V-D. Some conclusions reached as a result of the developments to date are given in Section V-E.

A. Current status

The PSL/PSA system described in this paper is operational on most larger computing environments which support interactive use, including IBM 370 series (OS/VS/TSO/CMS), Univac 1100 series (EXEC-8), CDC 6000/7000 series (SCOPE, TSS), Honeywell 600/6000 series (MULTICS, GCOS), AMDAHL 470/V8 (MTS), and PDP-10 (TOPS 10). Portability is achieved at a relatively high level; almost all of the system is written in ANSI Fortran.

PSL/PSA is currently being used by a number of organizations including AT&T Long Lines, Chase Manhattan Bank, Mobil Oil, British Railways, Petroleos Mexicanos, TRW Inc., the U.S. Air Force and others for documenting systems. It is also being used by academic institutions for education and research.

B. Benefit/cost analysis of computer-aided documentation

The major benefits claimed for computer-aided documentation are that the "quality" of the documentation is improved and that the cost of design, implementation, and maintenance will be reduced. The "quality" of the documentation, measured in terms of preciseness, consistency, and completeness is increased because the analysis must be more precise, the software performs checking, and the output reports can be reviewed for remaining ambiguities, inconsistencies, and omissions. While completeness can never be fully guaranteed, one important feature of the computer-aided method is that all the documentation that "exists" is the data base, and therefore the gaps and omissions are more obvious. Consequently, the organization knows what data it has, and does not have to depend on individuals who may not be available when a specific item of data about a system is needed. Any analysis performed and reports produced are up-to-date as of the time it is performed. The coordination among analysis is greatly simplified since each can work in his own area and still have the system specifications be consistent.

Development will take less time and cost less because errors, which usually are not discovered until programming or testing, have been minimized. It is recognized that one reason for the high cost of systems development is the fact that errors, inconsistencies, and omissions in specifications are frequently not detected until later stages of development: in design, programming, systems tests, or even operation. The use of PSL/PSA during the specification stage reduces the number of errors which will have to be corrected later. Maintenance costs are considerably reduced because the effect of a proposed change can easily be isolated, thereby reducing the probability that one correction will cause other errors.

The cost of using a computer-aided method during logical system design must be compared with the cost of performing the operations manually. In practice the cost of the various analyst functions of interviewing, recording, analyzing, etc., are not recorded separately. However, it can be argued that direct cost of documenting specifications for a proposed system using PSL/PSA should be approximately equal to the cost of producing the documentation manually. The cost of typing manual documentation is roughly equal to the cost of entering PSL statements into the computer. The computer cost of using PSA should not be more than the cost of analyst time in carrying out the analyses manually. (Computer costs, however, are much more visible than analysts costs.) Even though the total cost of logical system design is not reduced by using computer-aided methods, the elapsed time should be reduced because the computer can perform clerical tasks in a shorter time than analysis require.

C. Benefit/costs evaluation in practice

Ideally the adoption of a new methodology such as that represented by PSL/PSA should be based on quantitative evaluation of the benefits and costs. In practice this is seldom possible; PSL/PSA is no exception.

Very little quantitative information about the experience in using PSL/PSA, especially concerning manpower requirements and system development costs, is available. One reason for this lack of data is that the project has been concerned with developing the methodology and has not felt it necessary or worthwhile to invest resources in carrying out controlled experiments which would attempt to quantify the benefits. Furthermore, commercial and government organizations which have investigated PSL/PSA have, in some cases, started to use it without a formal evaluation; in other cases, they have started with an evaluation project. However, once the evaluation project is completed and the decision is made to use the PSL/PSA, there is little time or motivation to document the reasons in detail.

Organizations carrying out evaluations normally do not have the comparable data for present methods available and so far none have felt it necessary to run controlled experiments with both methods being used in parallel. Even

when evaluations are made, the results have not been made available to the project: because the organizations regard the data as proprietary.

The evidence that the PSL/PSA is worthwhile is that almost without exception the organizations which have seriously considered using it have decided to adopt it either with or without an evaluation. Furthermore, practically all organizations which started to use PSL/PSA are continuing their use (the exceptions have been caused by factors other than PSL/PSA itself) and in organizations which have adopted it, usage has increased.

D. Planned developments

PSL as a system description language was intended to be "complete" in that the logical view of a proposed information system could be described, i.e., all the information necessary for functional requirements and specifications could be stated. On the other hand, the language should not be so complicated that it would be difficult for analysis to use. Also, deliberately omitted from the language was any ability to provide procedural "code" so that analysis would be encouraged to concentrate on the requirements rather than on low-level flow charts. It is clear, however, that PSL must be extended to include more precise statements about logical and procedural information.

Probably the most important improvement in PSA is to make it easier to use. This includes providing more effective and simple data entry and modification commands and providing more help to the users. A second major consideration is performance. As the data base grows in size and the number of users increases, performance becomes more important. Performance is very heavily influenced by factors in the computing environment which are outside the control of PSA development. Nevertheless, there are improvements that can be made.

PSL/PSA is clearly only one step in using computer-aided methods in developing, operating, and maintaining information processing systems. The results achieved to date support the premise that the same general approach can successfully be applied to the rest of the system life cycle and that the data base concept can be used to document the results of the other activities in the system life cycle. The resulting data bases can be the basis for development of methodology, generalized systems, education, and research.

E. Conclusions

The conclusions reached from the development of PSL/PSA to date and from the effort in having it used operationally may be grouped into five major categories.

1) The determination and documentation of requirements and functional specifications can be improved by making use of the computer for recording and analyzing the collected data and statements about the proposed system.

2) Computer-aided documentation is itself a system of which the software is only a part. If the system is to be used, adequate attention must be given to the whole methodology, including: user documentation, logistics and mechanics of use, training, methodological support, and management encouragement.

3) The basic structure of PSL and PSA is correct. A system description language should be of the relational type in which a description consists of identifying and naming objects and relationships among them. The software system should be data-base oriented, i.e., the data entry and modification procedures should be separated from the output report and analysis facilities.

4) The approach followed in the ISDOS project has succeeded in bringing PSL/PSA into operational use. The same approach can be applied to the rest of the system life cycle. A particularly important part of this approach is to concentrate first on the documentation and then on the methodology.

5) The decision to use a computer-aided documentation method is only partly influenced by the capabilities of the system. Much more important are factors relating to the organization itself and system development procedures. Therefore, even though computer-aided documentation is operational in some organizations, that does not mean that all organizations are ready to immediately adopt it as part of their system life cycle methodology.

1. T. Aiken, "Initiating an Electronics Program," in *Proceedings of the 7th Annual Meeting of the Systems and Procedures Association* (1954).

2. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5 (May 1973), pp. 48-59.

3. _____, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," *Information Processing* (1974), pp. 192-97.

4. R.G. Canning, *Electronic Data Processing for Business and Industry* (New York: Wiley, 1956).

5. T.B. Glans, et al., *Management Systems* (New York: Holt, Rinehart, & Winston, 1968). Based on IBM's study Organization Plan, 1961.

6. J. Goldberg, ed., *Proceedings of the Symposium on High Cost of Software* (Stanford, Calif.: Stanford Research Institute, September 1973).

7. R.H. Gregory and R.L. Van Horn, *Automatic Data Processing Systems* (Belmont, Calif.: Wadsworth Publishing Co., 1960).

8. W. Hartman, H. Matthes, and A. Proeme, *Management Information Systems Handbook* (New York: McGraw-Hill, 1968).

9. E.A. Hershey and M. Bastarache, "PSA — Command Descriptions," University of Michigan, ISDOS Working Paper No. 91 (Ann Arbor, Mich.: 1975).

10. E.A. Hershey, et al., "Problem Statement Language — Language Reference Manual," University of Michigan, ISDOS Working Paper No. 68 (Ann Arbor, Mich.: 1975).

11. G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974).

12. *HIPO — A Design Aid and Documentation Technique*, IBM Corporation, Manual No. GC-20-1851 (White Plains, N.Y.: IBM Data Processing Division, October 1974).

13. M.N. Jones, "Using HIPO to Develop Functional Specifications," *Datamation*, Vol. 22, No. 3 (March 1976), pp. 112-25.

14. G.H. Larsen, "Software: Man in the Middle," *Datamation*, Vol. 19, No. 11 (November 1973), pp. 61-66.

15. G.J. Myers, *Reliable Software Through Composite Design* (New York: Petrocelli/Charter, 1975).

16. D.T. Ross and K.E. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 41-48.

17. H.W. Schrimpf and C.W. Compton, "The First Business Feasibility Study in the Computer Field," *Computers and Automation*, Vol. 18, No. 1 (January 1969), pp. 48-53.

18. D. Teichroew and M. Bastarache, "PSL User's Manual," University of Michigan, ISDOS Working Paper No. 98 (Ann Arbor, Mich.: 1975).

19. *Software Development and Configuration Management Manual*, TRW Systems Group, Manual No. TRW-55-73-07 (Redondo Beach, Calif.: December 1973).

20. U.S. Air Force, *Support of Air Force Automatic Data Processing Requirements Through the 1980's*, Electronics Systems Division, L.G. Hanscom Field, Report SADPR-85 (June 1974).

21. U.S. Department of Defense, *Automated Data Systems Documentation Standards Manual*, Manual 4120.17M (December 1972).

22. J.D. Warnier and B. Flanagan, *Entrainement de la Construction drs Programs D'Informatique*, Vols. I and II (Paris: Editions d'Organization, 1972).

23. N.E. Willworth, ed., *System Programming Management*, System Development Corporation, TM 1578/000/00 (Santa Monica, Calif.: March 13, 1964).

24. F.G. Withington, *The Organization of the Data Processing Function* (New York: Wiley Business Data Processing Library, 1972).

25. E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, N.J.: Prentice-Hall, 1979). (1979 edition of YOURDON's 1975 text.)

INTRODUCTION

DeMarco's "Structured Analysis and System Specification" is the final paper chosen for inclusion in this book of classic articles on the structured revolution. It is last of three on the subject of analysis, and, together with Ross/Schoman [Paper 22] and Teichroew/Hershey [Paper 23], provides a good idea of the direction that structured analysis will be taking in the next few years.

Any competent systems analyst undoubtedly could produce a five-page essay on "What's Wrong with Conventional Analysis." DeMarco, being an ex-analyst, does so with pithy remarks, describing conventional analysis as follows:

> "Instead of a meaningful interaction between analyst and user, there is often a period of fencing followed by the two parties' studiously ignoring each other. . . The cost-benefit study is performed backwards by deriving the development budget as a function of expected savings. (Expected savings were calculated by prorating cost reduction targets handed down from On High.)"

In addition to providing refreshing prose, DeMarco's approach differs somewhat — in terms of emphasis — from that of Teichroew/Hershey and of Ross/Schoman. Unlike his colleagues, DeMarco stresses the importance of the *maintainability* of the specification. Take, for instance, the case of one system consisting of six *million* lines of COBOL and written over a period of ten years by employees no longer with the organization. Today, *nobody knows what the system does!* Not only have the program listings and source code been lost — a relatively minor disaster that we all have seen too often — but the specifications are completely out of date. Moreover, the system has grown so large that neither the users nor the data processing people have the faintest idea of *what* the system is supposed to be doing, let alone *how* the mysterious job is being accomplished! The example is far from hypothetical, for this is the fate that all large systems eventually will suffer, unless steps are taken to keep the *specifications* both current and understandable across generations of users.

The approach that DeMarco suggests — an approach generally known today as structured analysis — is similar in form to that proposed by Ross and Schoman, and emphasizes a top-down, partitioned, graphic model of the system-to-be. However, in contrast to Ross and Schoman, DeMarco also stresses the important role of a *data dictionary* and the role of scaled-down specifications, or minispecs, to be written in a rigorous subset of the English language known as *Structured English*.

DeMarco also explains carefully how the analyst proceeds from a physical description of the user's current system, through a logical description of that same system, and eventually into a logical description of the new system that the user wants. Interestingly, DeMarco uses top-down, partitioned dataflow diagrams to illustrate this part of the so-called Project Life Cycle — thus confirming that such a graphic model can be used to portray virtually any system.

As in other short papers on the subject, the details necessary for carrying out DeMarco's approach are missing or are dealt with in a superficial manner. Fortunately, the details *can* be found: Listed at the end of the paper are references to three full-length books and one videotape training course, all dealing with the kind of analysis approach recommended by DeMarco.

DEMARCO

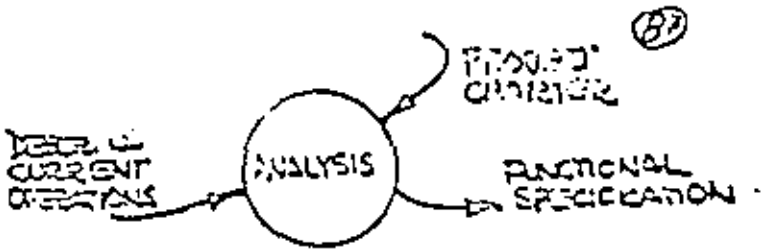
Structured Analysis and System Specification

When Ed Yourdon first coined the term Structured Analysis [1], the idea was largely speculative. He had no actual results of completed projects to report upon. His paper was based on the simple observation that some of the principles of top-down partitioning used by designers could be made applicable to the Analysis Phase. He reported some success in helping users to work out system details using the graphics characteristic of structured techniques.

Since that time, there has been a revolution in the methodology of analysis. More than 2000 companies have sent employees to Structured Analysis training seminars at YOURDON alone. There are numerous working texts and papers on the subject [2, 3, 4, 5]. In response to the YOURDON 1977 Productivity Survey [6, 7], more than one quarter of the respondents answered that they were making some use of Structured Analysis. The 1978 survey [8] shows a clear increase in that trend.

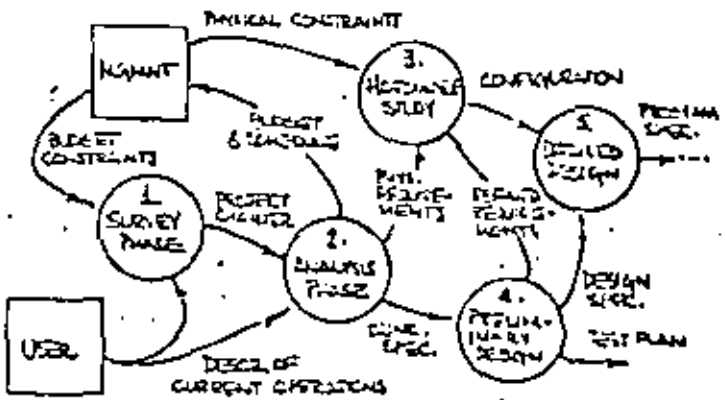
In this paper, I shall make a capsule presentation of the subject of Structured Analysis and its effect on the business of writing specifications of to-be-developed systems. I begin with a set of definitions:

1. What is analysis?



The analysis transformation.

Analysis is the process of transforming a stream of information about current operations and new requirements into some sort of rigorous description of a system to be built. That description is often called a Functional Specification or System Specification.



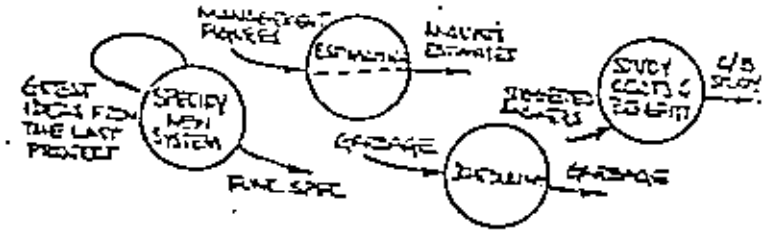
The Analysis Phase in context of the project life cycle.

In the context of the project life cycle for system development, analysis takes place near the beginning. It is preceded only by a Survey or Feasibility Study, during which a project charter (statement of changes to be considered, constraints governing development, etc.) is generated. The Analysis Phase is principally concerned with generating a specification of the system to be built.

But there are numerous required by-products of the phase, including budget, schedule and physical requirements information. The specification task is compounded of the following kinds of activities:

- user interaction,
- study of the current environment,
- negotiation,
- external design of the new system,
- I/O format design,
- cost-benefit study,
- specification writing, and
- estimating.

Well, that's how it works when it works. In practice, some of these activities are somewhat shortchanged. After all, everyone is eager to get on to the real work of the project (writing code). Instead of a meaningful interaction between analyst and user, there is often a period of fencing followed by the two parties' studiously ignoring each other. The study of current operations is frequently bypassed. ("We're going to change all that, anyway.") Many organizations have given up entirely on writing specifications. The cost-benefit study is performed backwards by deriving the development budget as a function of expected savings. (Expected savings were calculated by prorating cost reduction targets handed down from On High.) And the difficult estimating process can be conveniently replaced by simple regurgitation of management's proposed figures. So the Analysis Phase often turns out to be a set of largely disconnected and sometimes fictitious processes with little or no input from the user:



Analysis Phase activities.

2. What Is Structured Analysis?

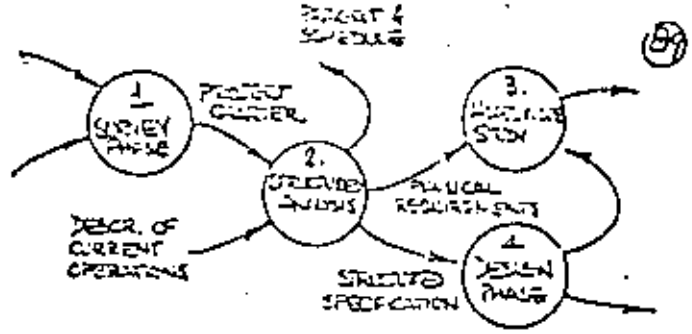


Figure 0: Structured Analysis in context of the project life cycle.

Structured Analysis is a modern discipline for conduct of the Analysis Phase. In the context of the project life cycle, its major apparent difference is its new principal product, called a Structured Specification. This new kind of specification has these characteristics:

- It is *graphic*, made up mostly of diagrams.
- It is *partitioned*, not a single specification, but a network of connected "mini-specifications."
- It is *top-down*, presented in a hierarchical fashion with a smooth progression from the most abstract upper level to the most detailed bottom level.
- It is *maintainable*, a specification that can be updated to reflect change in the requirement.
- It is *a paper model of the system-to-be;* the user can work with the model to perfect his vision of business operations as they will be with the new system in place.

I'll have more to say about the Structured Specification in a later section.

In the preceding figure, I have represented Structured Analysis as a single process (transformation of the project charter and description of current operations into outputs of budget, schedule, physical requirements and the Structured Specification). Let's now look at the details of that process:

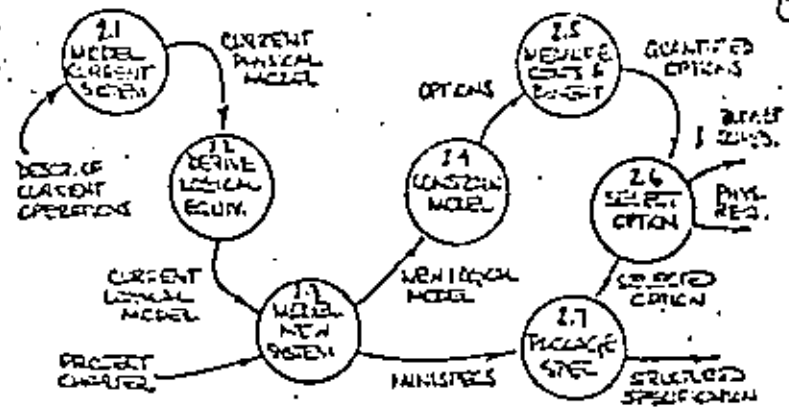


Figure 2: Details of Bubble 2, Structured Analysis.

Note that Figure 2 portrays exactly the same transformation as *Bubble 2* of the previous figure (same inputs, same outputs). But it shows that transformation in considerably more detail. It declares the component processes that make up the whole, as well as the information flows among them.

I won't insult your intelligence by giving you the thousand words that the picture in Figure 2 is worth. Rather than discuss it directly, I'll let it speak for itself. All that is required to complement the figure is a definition of the component processes and information flows that it declares:

Process 2.1. Model the current system: There is almost always a current system (system = integrated set of manual and perhaps automated processes, used to accomplish some business aim). It is the environment that the new system will be dropped into. Structured Analysis would have us build a paper model of the current system, and use it to perfect our understanding of the present environment. I term this model "physical" in that it makes use of the user's terms, procedures, locations, personnel names and particular ways of carrying out business policy. The justification for the physical nature of this first model is that its purpose is to be a verifiable representation of current operations, so it must be easily understood by user staff.

Process 2.2. Derive logical equivalent: The logical equivalent of the physical model is one that is divorced from the "hows" of the current operation. It concentrates instead on the "whats." In place of a description of a way to carry out policy, it strives to be a description of the policy itself.

Process 2.3. Model the new system: This is where the major work of the Analysis Phase, the "invention" of the new system, takes place. The project charter records the differences and potential differences between the current environment and the new. Using this charter and the logical model of the existing system, the analyst builds a new model, one that documents operations of the future environment (with the new system in place), just as the current model documents the present. The new model presents the system-to-be as a partitioned set of elemental processes. The details of these processes are now specified, one mini-spec per process.

Process 2.4. Constrain the model: The new logical model is too logical for our purposes, since it does not even establish how much of the declared work is done inside and how much outside the machine. (That is physical information, and thus not part of a logical model.) At this point, the analyst establishes the man-machine boundary, and hence the scope of automation. He/she typically does this more than once in order to create meaningful alternatives for the selection process. The physical considerations are added to the model as annotations.

Process 2.5. Measure costs and benefits: A cost-benefit study is now performed on each of the options. Each of the tentatively physicalized models, together with its associated cost-benefit parameters, is passed on in the guise of a "Quantified Option."

Process 2.6. Select option: The Quantified Options are now analyzed and one is selected as the best. The quanta associated with the option are formalized as budget, schedule, and physical requirements and are passed back to management.

Process 2.7. Package the specification: Now all the elements of the Structured Specification are assembled and packaged together. The result consists of the selected new physical model, the integrated set of mini-specs and perhaps some overhead (table of contents, short abstract, etc.).

3. What is a model?

The models that I have been referring to throughout are paper representations of systems. A system is a set of manual and automated procedures used to effect some business goal. In the convention of Structured Analysis, a model is made up of *Data Flow Diagrams* and a *Data Dictionary*. My definitions of these two terms follow:

The lack of feedback. Since pieces of Functional Specifications are unintelligible by themselves, we have nothing to show the user until the end of analysis. Our author-reader cycle may be as much as a year. There is no possibility of iteration (that is, frequently repeated attempts to refine and perfect the product). For most of the Analysis Phase, there is no product to iterate. The famous user-analyst dialogue is no dialogue at all, but a series of monologues.

The Classical Functional Specification is an unmaintainable monolith, the result of "communication" without benefit of feedback. No wonder it is *neither functional nor specific.*

5. How does Structured Analysis help?

I would not be writing this if I did not believe strongly in the value of Structured Analysis; I have been known to wax loquacious on its many virtues. But in a few words, these are the main ways in which Structured Analysis helps to resolve Analysis Phase problems:

- It attacks the problem of largeness by *partitioning.*
- It attacks the many problems of communication between user and analyst by *iterative communication* and an *inversion of viewpoint.*
- It attacks the problem of specification maintenance by *limited redundancy.*

Since all of these ideas are rather new in their application to analysis, I provide some commentary on each:

The concept of *partitioning* or "functional decomposition" may be familiar to designers as the first step in creating a structured design. Its potential value in analysis was evident from the beginning — clearly, large systems cannot be analyzed without some form of concurrent partitioning. But the direct application of Design Phase functional decomposition tools (structure charts and HIPO) caused more problems than it solved. After some early successful experiments [9], negative user attitudes toward the use of hierarchies became apparent, and the approach was largely abandoned. Structured Analysis teaches use of leveled Data Flow Diagrams for partitioning, in place of the hierarchy. The advantages are several:

- The appearance of a Data Flow Diagram is not at all frightening. It seems to be simply a picture of the subject matter being discussed. You never have to explain an arbitrary convention to the user — you don't explain anything at all. You simply use the diagrams. I did precisely that with you in this paper; I used Data Flow Diagrams as a descriptive tool, long before I had even defined the term.
- Network models, like the ones we build with Data Flow Diagrams, are already familiar to some users. Users may have different names for the various tools used — Petri Networks or Paper Flow Charts or Document Flow Diagrams — but the concepts are similar.
- The act of partitioning with a Data Flow Diagram calls attention to the interfaces that result from the partitioning. I believe this is important, because the complexity of interfaces is a valuable indicator of the quality of the partitioning effort: the simpler the interfaces, the better the partitioning.

I use the term *iterative communication* to describe the rapid two-way interchange of information that is characteristic of the most productive work sessions. The user-analyst dialogue has got to be a dialogue. The period over which communication is turned around (called the author-reviewer cycle) needs to be reduced from months to minutes. I quite literally mean that fifteen minutes into the first meeting between user and analyst, there should be some feedback. If the task at hand requires the user to describe his current operation, then fifteen minutes into that session is not too early for the analyst to try telling the user what he has learned. "OK, let me see if I've got it right. I've drawn up this little picture of what you've just explained to me, and I'll explain it back to you. Stop me when I go wrong."

Of course, the early understanding is always imperfect. But a careful and precise declaration of an imperfect understanding is the best tool for refinement and correction. The most important early product on the way to developing a good product is an imperfect version. The human mind is an iterative processor. It never does anything precisely right the first time. What it does consummately well is to make a slight improvement to a flawed product. This it can do again and again. The idea of developing a flawed early version and then refining and refining to make it right is a very old one. It is called *engineering.* What I am proposing is an engineering approach to analysis and to the user-analyst interface.

The product that is iterated is the emerging system model. When the user first sees it, it is no more than a rough drawing made right in front of him during the discussion. At the end, it is an integral part of the Structured Specification. By the time he sees the final specification, each and every page of it should have been across his desk a half dozen times or more.

I mentioned that Structured Analysis calls for an *inversion of viewpoint.* This point may seem obscure because it is not at all obvious what the viewpoint of classical analysis has been. From my reading of hundreds of Classical Func-

tional Specifications over the past fifteen years, I have come to the conclusion that their viewpoint is most often that of the computer. The classical specification describes what the computer does, in the order that it does it, using terms that are relevant to computers and computer people. It frequently limits itself to a discussion of processing inside the machine and data transferred in and out. Almost never does it specify anything that happens outside the man-machine boundary.

The machine's viewpoint is natural and useful for those whose concerns lie inside (the development staff), and totally foreign to those whose concerns lie outside (the user and his staff). Structured Analysis adopts a different viewpoint, that of the data. A Data Flow Diagram follows the data paths wherever they lead, through manual as well as automated procedure. A correctly drawn system model describes the outside as well as the inside of the automated portion. In fact, it describes the automated portion *in the context of the complete system*. The viewpoint of the data has two important advantages over that of any of the processors:

- The data goes everywhere, so its view is all-inclusive.
- The viewpoint of the data is common to those concerned with the inside as well as those concerned with the outside of the man-machine boundary.

The use of *limited redundancy* to create a highly maintainable product is not new. Any programmer worth his salt knows that a parameter that is likely to be changed ought to be defined in one place and one place only. What is new in Structured Analysis is the idea that the specification ought to be maintainable at all. After all, aren't we going to "freeze" it?. If we have learned anything over the past twenty years, it is that the concept of freezing specifications is one of the great pipe dreams of our profession. Change cannot be forestalled, only ignored. Ignoring change assures the building of a product that is out of date and unacceptable to the user. We may endeavor to hold off selected changes to avoid disruption of the development effort, but we can no longer tolerate being *obliged* to ignore change just because our specification is impossible to update. It is equally intolerable to accept the change, without updating the specification. The specification is our mechanism for keeping the project on target, tracking a moving goal. Failure to keep the specification up to date is like firing away at a moving target with your eyes closed. A key concept of Structured Analysis is development of a specification with little or no redundancy. This is a serious departure from the classical method that calls for specification of everything at least eleven times in eleven places. Reducing redundancy, as a by-product, makes the resultant specification considerably more concise.

6. What is a Structured Specification?

Structured Analysis is a discipline for conduct of the Analysis Phase. It includes procedures, techniques, documentation aids, logic and policy description tools, estimating heuristics, milestones, checkpoints and by-products. Some of these are important, and the rest merely convenient. What is most important is this simple idea: Structured Analysis involves building a new kind of specification, a Structured Specification, made up of Data Flow Diagrams, Data Dictionary and mini-specs.

The roles of the constituent parts of the Structured Specification are presented below:

The *Data Flow Diagrams* serve to partition the system. The system that is treated by the Data Flow Diagram may include manual as well as automated parts, but the same partitioning tool is used throughout. The purpose of the Data Flow Diagram is not to specify, but to declare. It declares component processes that make up the whole, and it declares interfaces among the components. Where the target system is large, several successive partitionings may be required. This is accomplished by lower-level Data Flow Diagrams of finer and finer detail. All the levels are combined into a leveled DFD set.

The *Data Dictionary* defines the interfaces that were declared on the Data Flow Diagrams. It does this with a notational convention that allows representation of dataflows and stores in terms of their components. (The components of a dataflow or store may be lower-level dataflows, or they may be data elements.) Before the Structured Specification can be called complete, there must be one definition in the Data Dictionary for each dataflow or data store declared on any of the Data Flow Diagrams.

The *mini-specs* define the elemental processes declared on the Data Flow Diagrams. A process is considered elemental (or "primitive") when it is not further decomposed into a lower-level Data Flow Diagram. Before the Structured Specification can be called complete, there must be one mini-spec for each primitive process declared on any of the Data Flow Diagrams.

The YOURDON Structured Analysis convention includes a set of rules and methods for writing mini-specs using Structured English, decision tables and trees and certain non-linguistic techniques for specification. For the purposes of this paper, such considerations are at the detail level — once you have partitioned to the point at which each of the mini-specs can be written in a page or less, it doesn't matter too terribly much how you write them.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

TEMA III

DISEÑO ESTRUCTURAL

ING. ALEJANDRO ACOSTA

MARZO, 1982

Contenido

Contenido

1.1. Objetivos del Diseño Estructurado

Existen una serie de problemas comunes a la mayoría de los proyectos de desarrollo de software, los que a continuación se mencionan se deben a la falta de técnicas apropiadas para su desarrollo:

1. Difíciles de Administrar.

La principal consecuencia de este problema es que cuando se detecta un retraso o una desviación en lo presupuestado, es demasiado tarde para corregirlo. Además los soluciones convencionales aplicables a proyectos de otras áreas no son aplicables al desarrollo de Software, por ejemplo, añadir recursos humanos tiene un efecto que se puede predecir con la ley de Brooks (1):

> Adding manpower to a late Software project makes it later.

2. Poco Satisfactorio.

Comparado con otras disciplinas de Ingeniería, el desarrollo de Software es poco profesional, frecuentemente los sistemas terminados dejan pocas satisfacciones a usuarios y diseñadores y muchas frustraciones. Esto lleva a que el desarrollo de Software sea, en general, una actividad poco productiva.

3. Poco Confiable.

Es muy frecuente que una vez que el sistema se "libera", falle una y otra vez. Las fallas pueden no tener razones obvias o peor, se pueden producir resultados incorrectos que pasen inadvertidos.

4. Inflexible.

Cuando un sistema llega a trabajar se considera un milagro. Quién supone que se podrán realizar futuros cambios sin problema?

5. Difícil de Mantener.

Para modificar un sistema, se requiere:

- Entender cómo trabaja el sistema actual.
- Concebir el cambio.
- Calcular las ramificaciones del cambio.

6. Ineficientes.

La idea generalizada de eficiencia está influenciada por la época en que el costo del procesador central era muy superior a los otros costos, de aquí que se considere eficientes a los programas que optimizan el uso del procesador central en lugar de aquellos que hacen uso eficiente de recursos escasos.

Con el Diseño Estructurado se pretende hacer sistemas cuyas características eviten los problemas antes mencionados (y otros no mencionados).

El objetivo del Diseño Estructurado es hacer Sistemas que sean:

- Utiles
- Mantenibles
- Modificables
- Flexibles
- Eficientes
- Generales

Todos estos elementos son componentes de un objetivo de calidad.

1.2 Filosofía General.

Podemos resumir los objetivos del Diseño Estructurado en uno solo: Producir sistemas económicamente convenientes (baratos).

Un diseño exitoso se basa en un principio conocido desde los días de Julio César:

- Divide y vencerás.

Veamos como este principio se aplica a los siguientes aspectos:

1. Implementación.

   El costo de implementar sistemas de cómputo será minimizado cuando las partes del problema sean:

   - suficientemente pequeñas
   - solucionables separadamente

2. Mantenimiento.

   De manera similar, el costo de mantenimiento será minimizado cuando las partes del sistema sean:

   - fácilmente relacionados a la aplicación
   - suficientemente pequeñas
   - corregibles separadamente

3. Modificaciones.

   Finalmente, el costo de modificar un sistema será minimizado cuando sus partes sean:

   - fácilmente relacionadas al problema
   - modificables separadamente

En resumen, podemos establecer la siguiente filosofia:

> Implementación, Mantenimiento y Modificación serán generalmente minimizados cuando cada parte del sistema corresponda a exactamente una parte bien definida y pequeña del problema, y cada relación entre partes del sistema corresponda solo a relaciones entre partes del problema.

Esto significa que un buen diseño es un ejercicio consistente en dividir y organizar las partes de un sistema.

2.0 LAS IDEAS COMPLEMENTARIAS

Para alcanzar los objetivos del diseño estructurado, se requiere:

- Una herramienta de modelado para planear el sistema.
- Alguna manera para controlar la complejidad de sistemas no triviales.

2.1 El Modelo.

Un modelo es simplemente un cuerpo ordenado de hipótesis acerca de un sistema complejo; es un intento por entender algún aspecto de la infinita variedad de ellos que presenta el mundo, seleccionando o partir de percepciones y de experiencias pasadas, un cuerpo de observaciones generales aplicables al problema en cuestión.

Los modelos se presentan de varias maneras como son gráficas, ecuaciones, modelos a escala, maquetas, simuladores, etc. Los modelos son importantes en las disciplinas de ingeniería y su selección depende del problema en cuestión.

Para alcanzar los objetivos del Diseño Estructurado, requerimos de un modelo con las siguientes características:

- Gráfico.
- Divisible (top-down).
- Riguroso.
- Capaz de predecir el comportamiento del sistema.
- Que sea una consecuencia natural del Análisis Estructurado.
- Que sea una entrada natural para la Implementación Estructurada.
- Documentación básica del sistema.
- Ayuda para mantener y/o modificar el sistema.

2.2 El Control de Complejidad.

> The basic pattern of my approach will be to compose the program in minute steps, deciding each time as little as posible. As the problem analysis proceeds, so does the further refinement of my program. E. W. DIJKSTRA

La herramienta más útil para el control de complejidad en el diseño de sistemas es la Caja Negra.

Una Caja Negra es un mecanismo del cual se conoce:

- Sus entradas
- Sus salidas
- Lo que hace
- No se necesita saber cómo lo hace

Cuando los diseños se hacen con cajas negras pequeñas e independientes éstas son fácilmente entendidas, probadas, corregidas, mantenidas y modificadas..

El uso de la Caja Negra para controlar la complejidad del sistema, radica en dividir el sistema en Cajas Negras conectadas de tal forma que:

- Cada Caja Negra corresponda a una parte bien definida del problema.
- Cada Caja negra sea fácil de entender.
- Las conecciones entre Cajas Negras correspondan a conecciones del problema.
- Las conexiones entre Cajas Negras sean lo más simple posible, de tal manera que las Cajas Negras sean independientes entre sí.

2.3 Las Herramientas.

El Diseño Estructurado se apoya en el uso de dos herramientas:

- El Diagrama de Estructura.
- El Diagrama de Datos.

. El Diagrama de Estructura muestra la división del sistema, su jerarquía y su organización.

. El Diagrama de Datos muestra la división del sistema, sus flujos de datos y su organización.

Existen otras herramientas complementarias a las antes mencionadas como son las heurísticas de diseño, la morfología del sistema, criterios de transformación, etc., las cuales intervienen en el Diseño Estructurado; sin embargo la aplicación de estas herramientas complementarias requiere del Diagrama de Datos y del Diagrama de Estructura.

2.4 La Metodología.

Es posible dividir el proceso de diseño en:

- Diseño general
- Diseño detallado

El diseño general consiste en decidir qué funciones, parámetros y relaciones son requeridas para el programa (o sistema). - Diseño detallado es cómo implementar los funciones.

En términos muy generales, la metodología del Diseño Estructurado consiste en la aplicación sistemática de las herramientas de diseño de la siguiente forma:

- Hacer el Diagrama de Datos.
- Aplicando criterios de transformación, hacer el Diagrama de Estructura.
- Factorizar el Diagrama de Estructura.
- Analizar el Diagrama de Estructura utilizando criterios de:
  - Acoplamiento
  - Cohesión
  - Heurísticos de diseño
- Especificar cada módulo

3.0 EL DIAGRAMA DE ESTRUCTURA

El Diagrama de Estructura es una representación gráfica del sistema que se utiliza como herramienta para el diseño, implementación, documentación, modificación y mantenimiento del sistema. Es una herramienta, no un método.

El Diagrama de Estructura es un modelo independiente del tiempo de las relaciones jerárquicas de los módulos de un programa o sistema; es por esto que no se puede inferir de un Diagrama de Estructura, cuál es el orden en que se ejecutan los módulos.

Los elementos que forman el Diagrama de Estructura son los siguientes:

- Módulos (cuadros).
- Conexiones (flechas)
- Interfases (Nombres de datos que entran y salen de cada módulo)

Un Módulo es una secuencia de instrucciones continuas de programa, confinadas por variables limítrofes, tienen un identificador.

Los Módulos tienen los siguientes atributos:

El qué:
- Entradas (lo que obtiene de su invocador)
- Salidas (lo que regresa a su invocador)
- Función (lo que hace a los entradas para producir las salidas)

El cómo:
- Mecánica (cómo hace su función)
- Datos internos (espacio privado de trabajo, solo él los referencia)

Nótese que no existe diferencia entre Módulo, Programa y Sistema.

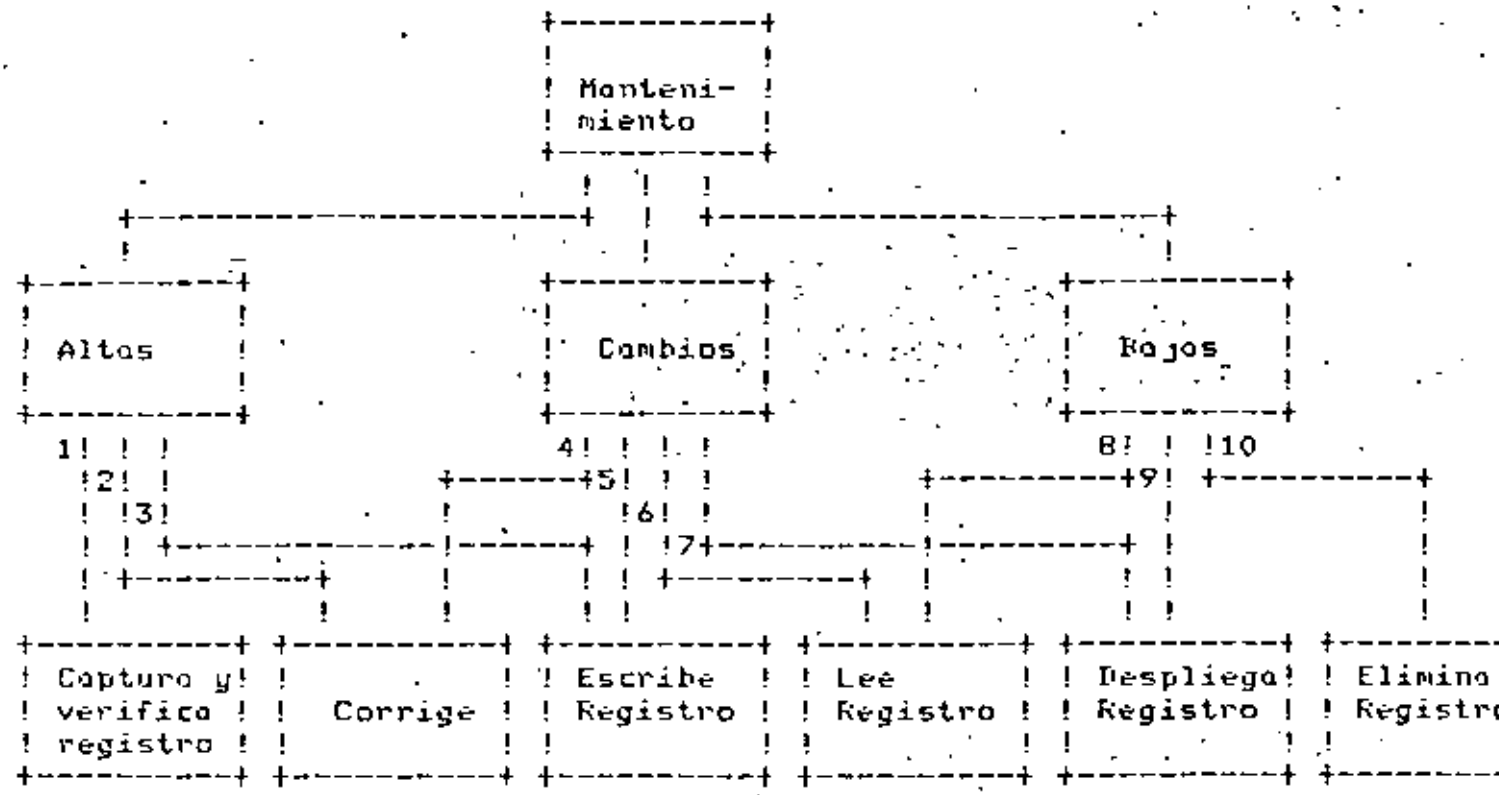
El Diagrama de Estructura muestra lo siguiente:

- La división del sistema en Módulos
- Jerarquía y organización de los Módulos
- Interfases de comunicación entre Módulos
- Nombres (funciones) de los Módulos

El Diagrama de Estructura no muestra:

- Mecanismos internos de los Módulos
- Datos internos de los Módulos

Ejemplo de Diagrama de Estructura:



	Entran (al subordinado)	Salen (del subordinado)
1		Registro
2	Registro	Registro
3	Registro	
4	Registro	Registro
5	Registro	
6		Registro
7	Registro	
8		Registro
9	Registro	
10	Registro	Resultado

4.0 ACOPLAMIENTO

Dos Módulos son totalmente independientes si cada uno puede funcionar completamente sin la presencia del otro. Esta definición implica que no hay interconecciones directas o indirectas, explícitas o implícitas, obvias u oscuras, entre los Módulos. Esto marca el punto cero en la escala de "dependencia" entre Módulos.

En general, entre más interconecciones existan entre Módulos, serán más dependientes entre si.

El Acoplamiento es una medida de la interdependencia de un Módulo respecto a otro. Entonces, los Módulos altamente acoplados están unidos por interconecciones rígidos. Los Módulos holgadamente acoplados estan unidos por interconecciones débiles. Los Módulos no acoplados son aquellos que no tienen interconecciones.

El Acoplamiento es el criterio más importante para juzgar las bondades de un diseño.

4.1 Factores que intervienen en el Acoplamiento

Existen cuatro factores principales que pueden incrementar o decrementar el acoplamiento de módulos, son los siguientes:

1. Tipo de conexión entre módulos.

   Existen tres tipos de sistemas:

   - Conexiones mínimas.
   - Conexiones normales.
   - Conexiones patológicas.

2. Complejidad de la interfase.

   Se puede aproximar por el número de elementos pasados entre los módulos, entre más elementos haya, la interfase será más compleja.

3. Tipo de flujo de información a lo largo de la conexión.

   Los sistemas con acoplamiento de datos tienen menor acoplamiento que los de acoplamiento de control y éstos a su vez son mejores que los de acoplamiento híbrido.

4. Momento de unión de la conexión.

   Las conexiones unidas a referencias fijas al momento de ejecución, tienen menor acoplamiento que cuando la unión se efectúa al tiempo de cargo o de compilación o de codificación.

El concepto de Acoplamiento invita al desarollo de un concepto nuevo: Desacoplamiento.

Desacoplamiento es cualquier técnico o método sistemático para volver a los módulos más independientes.

4.2 Tipos de acoplamiento

Entre menor sea el acoplamiento entre cualesquiera dos módulos, éstos serán más independientes y el diseño será mejor.

Existen cinco tipos de acoplamiento:

- Datos
- Estampado
- Control
- Area común
- Contenido

El mejor acoplamiento es el de datos y el peor el de contenido.

Si se presenta mas de un tipo de acoplamiento entre módulos, es el peor el que aplica.

4.2.1 Acoplamiento de Datos. -

Acoplamiento de datos es cuando solo los datos necesarios son comunicados entre módulos.

Este tipo de acoplamiento es el más deseable, y de hecho, cualquier sistema puede construirse de tal manera que el único acoplamiento sea de datos.

4.2.2 Acoplamiento de Estampado. -

Dos módulos presentan acoplamiento de estampado si referencian la misma estructura de datos (no global).

Una estructura de datos es un compuesto de elementos.

Este tipo de acoplamiento presenta el siguiente problema: un cambio en la estructura de datos afectará a todos los módulos que están "estampados" con la estructura;

sin embargo, usando una buena y natural estructura de datos, este tipo de acoplamiento se acerca al acoplamiento de datos.

4.2.3 Acoplamiento de Control. -

Dos módulos presentan acoplamiento de control si se comunican usando al menos un elemento de control.

Este acoplamiento es indeseable porque uno o ambos módulos no serán una caja negra.

Un peligro relacionado con el acoplamiento de control es el denominado "acoplamiento hibrido".

Acoplamiento hibrido sucede cuando se tienen dos o mas significados para el mismo dato.

4.2.4 Acoplamiento de Area Común. -

Un grupo de módulos presentan acoplamiento de área común si comparten una misma área global de datos.

Existen varios problemas con el acoplamiento de control:

1. Es más dificil reusar los módulos que utilizan áreas comunes ya que usualmente lo hacen por su nombre.

2. El mantenimiento se hace más dificil sobre todo cuando se pasan diferentes tipos de datos através del área común.

3. El uso de áreas globales dificulta la legibilidad de los programas.

4. Es difícil saber qué módulos usan qué datos.

5. En algunos lenguajes como FORTRAN, donde la posición de los datos en COMMON es importante, además del problema de acoplamiento de área común, se tiene un problema de acoplamiento de estampado.

6. Un error en cualquier módulo usando el área común puede aparecer en otro módulo (que tambien use el área común) ya que el área común no está protegida por ningún módulo.

4.2.5 Acoplamiento de Contenido. -

Este es el peor caso, ocurre cuando:

1. Un módulo altera instrucciones en otro módulo.

2. Un módulo referencia o cambia datos contenidos en otro módulo

3. Un módulo brinca a otro.

4. Dos módulos comparten los mismos literales.

Los casos 1, 2 y 3 también se conocen como Patológicos.

El problema principal es que un cambio pequeño y de apariencia inocente o uno de los módulos puede desquiciar o otro módulo en cualquier parte del sistema.

5.0 COHESION

Cohesión es una medida de la consistencia de la asociación de los elementos dentro de un módulo.

Un elemento es:

- Una instrucción
- Un grupo de instrucciones
- Una llámada a otro módulo

Es deseable tener módulos fuertes, altamente cohesivos, es decir, módulos cuyos elementos están altamente relacionados.

Claramente, cohesión y acoplamiento estan relacionados. A mayor cohesión de los módulos del sistema, existirá el menor acoplamiento.

5.1 Tipos de Cohesión.

La cohesión es una segunda medida de que tan bien dividimos el sistema. Existen los siguientes tipos:

- Funcional
- Secuencial
- Comunicacional
- De procedimiento
- Temporal
- Lógico
- Coincidental

Siendo mejor la cohesión funcional y peor lo coincidental.

5.1.1 Cohesión Funcional. -

Un módulo con cohesión funcional es aquel en el que todos sus elementos contribuyen a uno y solo una tarea completa; cada elemento es parte integral y es esencial para la ejecución de la función del módulo.

5.1.2 Cohesión Secuencial. -

Un módulo con cohesión secuencial es aquel en el que sus elementos están involucrados en tareas en las que datos que salen de un elemento sirven de entrada a otro elemento.

Los módulos con cohesión secuencial son fuertes, usualmente tienen buen acoplamiento. No son ideales porque contienen varias funciones que no forman una sola función completa.

5.1.3 Cohesión Comunicacional. -

Un módulo con cohesión comunicacional es aquel en el que sus elementos contribuyen a diferentes tareas, pero cada tarea se refiere a los mismos parámetros de entrada y/o salida.

Este tipo de cohesión es fuerte ya que el agrupamiento de funciones dentro de un módulo tienen alguna relación con el problema en lugar de la implementación de la solución.

Al dividir estos módulos en módulos separados, se simplifica el acoplamiento y se incrementa la cohesión.

5.1.4 Cohesión de Procedimiento. -

Un módulo con cohesión de procedimiento es aquel en el que el control fluye de un elemento al siguiente, pero, los datos no necesariamente fluyen de la misma manera.

El mayor problema con estos módulos es que manipulan resultados parciales, variables internas, banderas, switches, etc. y que tienen partes de varios funciones.

5.1.5 Cohesión Temporal. -

Los módulos con cohesión temporal son aquellos cuyos elementos están relacionados en tiempo.

Usualmente estos elementos realmente pertenecen a diferentes funciones.

5.1.6 Cohesión Lógica. -

Los módulos con cohesión lógica son aquellos en que sus elementos aparentan estar relacionados a tareas de la misma categoría general (debería llamarse cohesión ilógica).

5.1.7 Cohesión Coincidental. -

Un módulo con cohesión coincidental, tiene elementos sin relación significativa entre ellos. Usualmente ejecutan tareas diferentes y sin relación para diferentes "jefes".

6.0 ANALISIS DE TRANSFORMACION.

Análisis de transformación es un de los estrategias principales para diseñar sistemas altamente balanceados. Tambien se le conoce como Diseño de la Transformación central.

Para el Análisis de transformación se requiere del Diagrama de Datos. El resultado es un Diagrama de Estructura en el que el módulo superior trabaja con datos altamente procesados, datos lógicos.

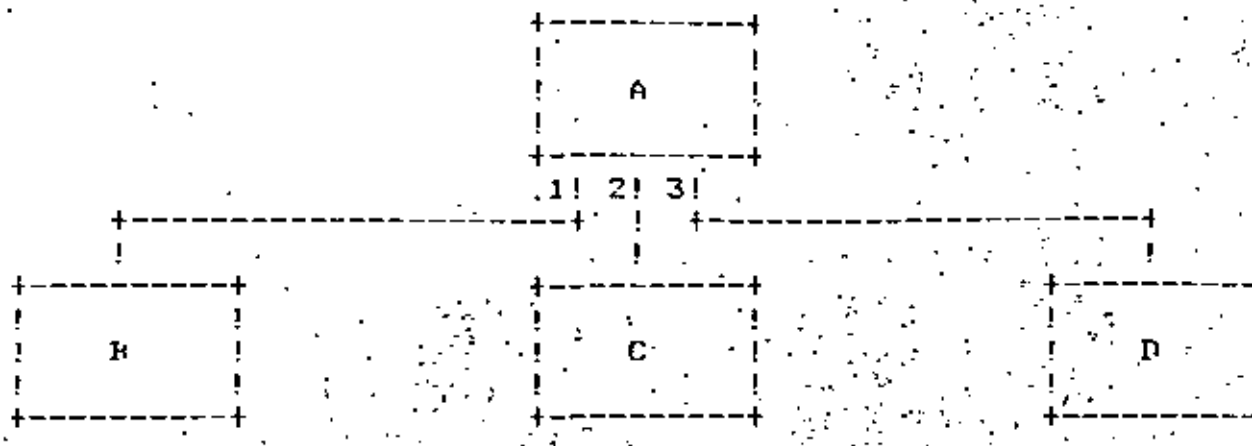
6.1 La estrategia.

1. Dibujar el diagrama de datos del problema.

2. Identificar la transformación central. Esto puede realizarse siguiendo las ramas aferentes y eferentes del diagrama hasta encontrar los puntos en que los datos son independientes de los dispositivos de entrada-salida. Los procesos entre estos puntos forman la transformación central.

3. Identificar las ramas principales de datos de entrada y salida. Determinar los puntos de mayor abstracción.

4. Diseñe la estructura a partir de la información previa con un módulo para cada rama principal de entrada y un modulo para cada rama principal de salida. En general se llegará a la estructura siguiente:



	Entran (al subordinado)	Salen (del subordinado)
1	Usualmente nada	Datos abstractos de entrada
2	Datos abstractos de entrada	Datos abstractos de salida
3	Datos abstractos de salida	Usualmente nada

5. Para cada uno de los módulos de entrada, identificar la última transformación necesaria para producir los datos en la forma en que los regresa el módulo. Después identifique la forma de la entrada justo antes de la última transformación. Para módulos de salida, identifique el primer proceso necesario para acercarse a la salida deseada con el formato deseado. Repita este paso hasta llegar a la forma original de los datos y el formato deseado de los resultados.

7.0 ANALISIS DE TRANSACCIONES

Una transacción es cualquier elemento de datos, control, señal, evento, o cambio de estado que causa, dispara o inicia alguna acción o secuencia de acciones.

En otras palabras, una transacción es una parte de datos que puede ser cualquiera de un número de tipos, cada tipo requiere diferente procesamiento.

La estrategia de análisis de transacciones, simplemente reconoce que los diagramas de datos de este tipo pueden mapearse a una estructura modular particular. El centro de transacciones de un sistema, debe ser capaz de:

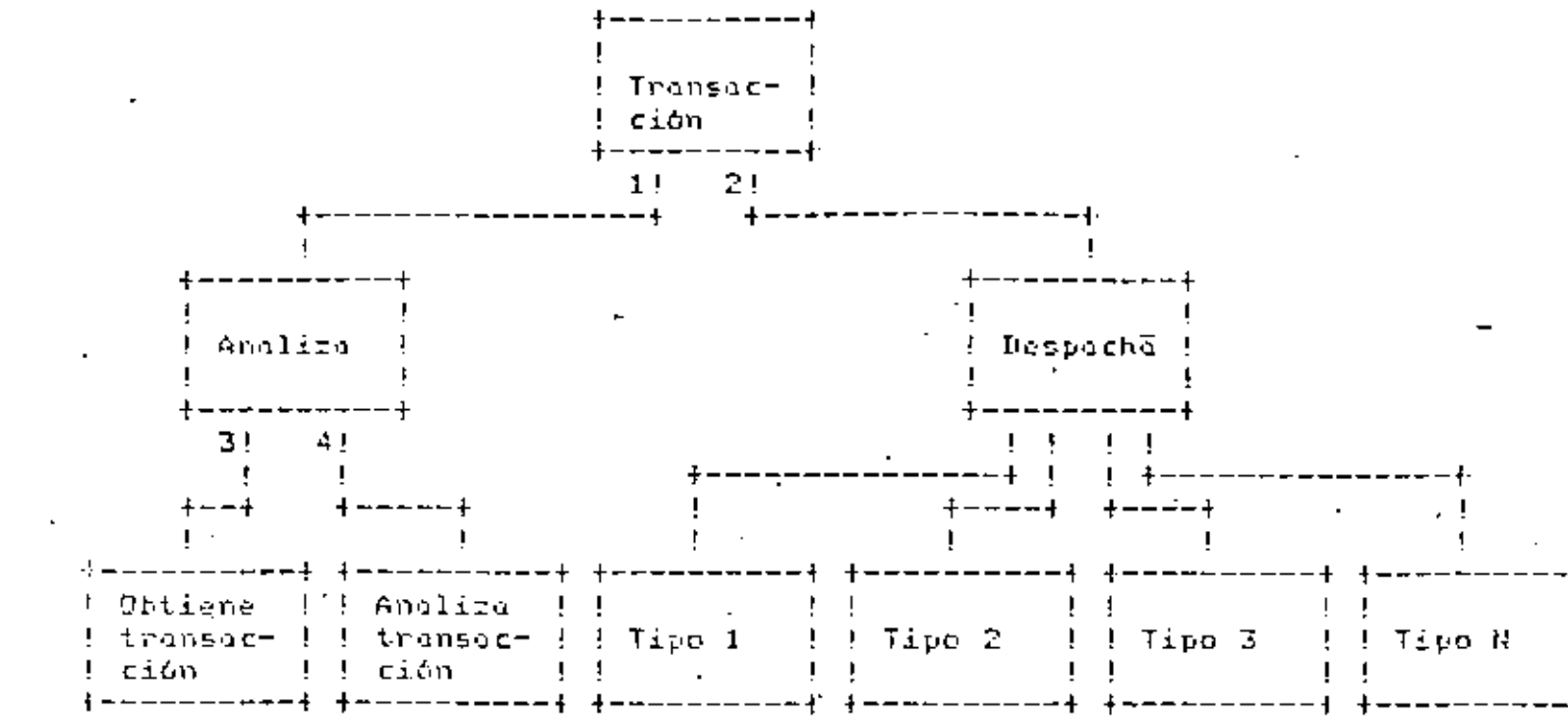
1. Obtener (responder a) las transacciones en su forma mas primitiva.

2. Analizar cada transaccion para determinar su tipo.

3. Despachar dependiendo de la transacción.

4. Completar el proceso de cada transacción.

En su forma más factorizada, el centro de transacciones puede ser modularizado de la siguiente forma:



	Entran (al subordinado)	Salen (del subordinado)
1		Transacción codificado internamente
2	Transacción codificado internament	
3		Transacción
4	Transacción	Transacción analizado y codificado

7.1 La estrategia.

1. Identificar los orígenes de transacciones.

2. Especificar la organización de transacciones apropiada (la figura anterior es un buen modelo en general).

3. Identificar la transaccion y las acciones que la definen.

4. Note situaciones poteciales en que se puedan combinar módulos.

5. Para cada transacción o grupo cohesivo de transacciones, especificar un módulo de transacciones para procesarlo completamente.

6. Para cada acción en una transacción, especificar un módulo de acción subordinado al correspondiente módulo de transacción.

7. Para cada paso detallado en un módulo de acción, especificar un módulo detallado subordinado al módulo de acción que lo necesite.

7.1 La estrategia.

1. Identificar los orígenes de transacciones.

2. Especificar la organización de transacciones apropiada (la figura anterior es un buen modelo en general).

3. Identificar la transaccion y las acciones que la definen.

4. Note situaciones poteciales en que se puedan combinar módulos.

5. Para cada transacción o grupo cohesivo de transacciones, especificar un módulo de transacciones para procesarlo completamente.

6. Para cada acción en una transacción, especificar un módulo de acción subordinado al correspondiente módulo de transacción.

7. Para cada paso detallado en un módulo de acción, especificar un módulo detallado subordinado al módulo de acción que lo necesite.

8.0 CASOS PARTICULARES

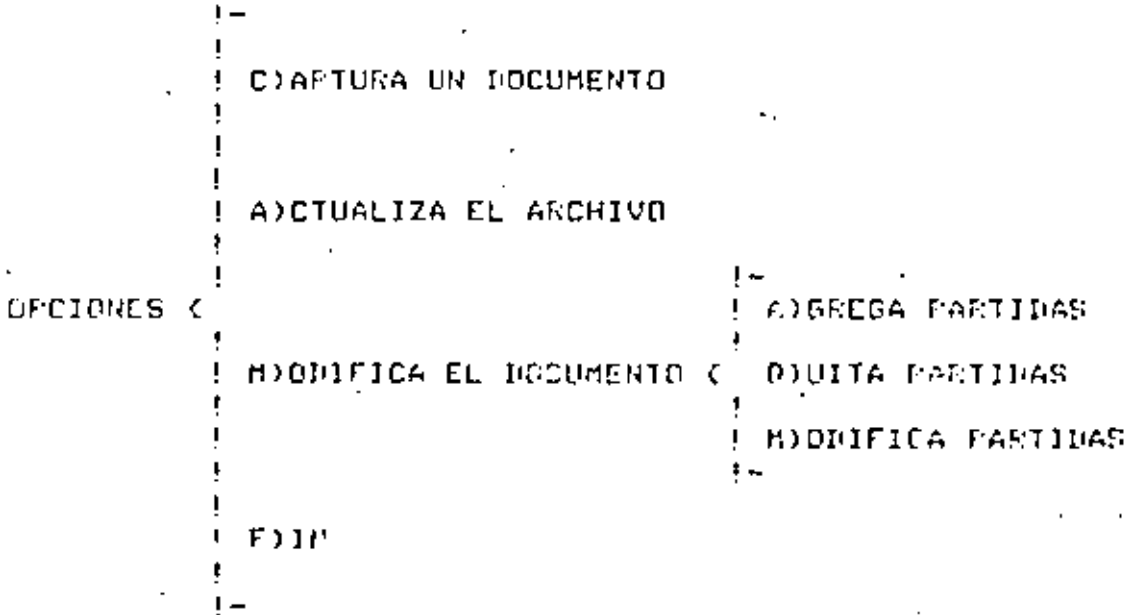
8.1 Programas interactivos

Un programa interactivo tiene los siguientes características:

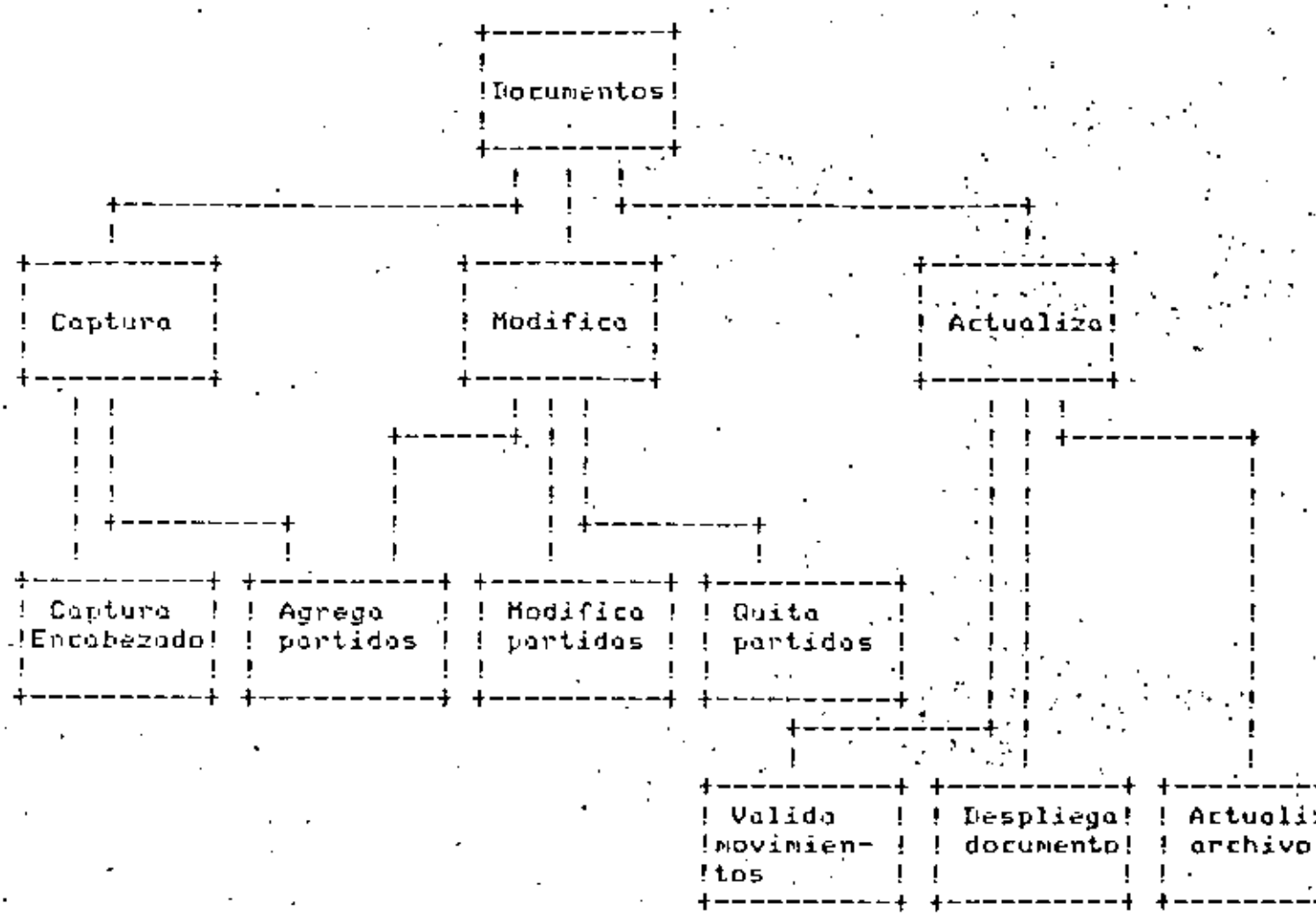
1. Conversacional...
2. Estructurado por tipo función.
3. Cada transacción es validado (consultando archivos) y actualizado instantaneamente.

Estos programas por naturaleza presentan una estructura arborea. Por tanto, la estrategia más apropiada para diseñarlos es a partir del árbol de funciones.

Ejemplo de Arbol de funciones:



Ejemplo de estructura a partir del árbol de funciones.



8.2 Programas Batch.

Un programa batch tiene las siguientes caracteristicas:

1. Controlado mediante comandos. Si el comando es incorrecto, se requiere realimentarlo hasta que sea correcto.

2. No es posible hacer correcciones a los errores de validación al momento de ejecutar el programa.

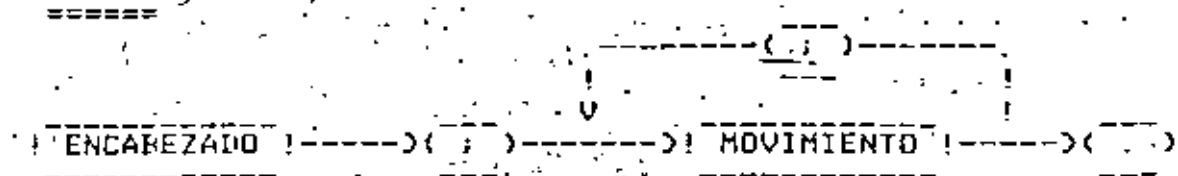
3. Tipicamente se utilizan en procesos periódicos.

La manera más adecuada para diseñar estos programas es a partir de una definición sintáctica formal de los comandos y de la información que se va a procesar.

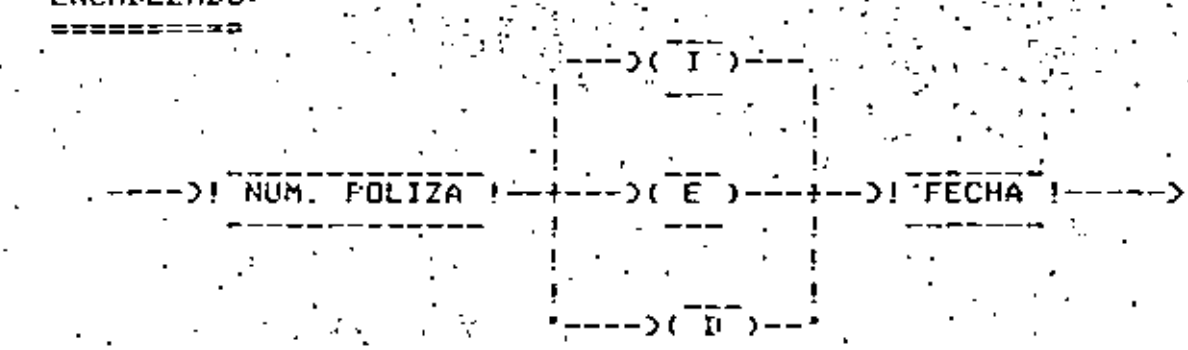
Una de las definiciones sintácticas más convenientes es la notación de Backus' Naur (BNF) la cual se puede representar gráficamente con los diagramas de ferrocarril (estilo PASCAL).

Ejemplo de diagrama de sintaxis:

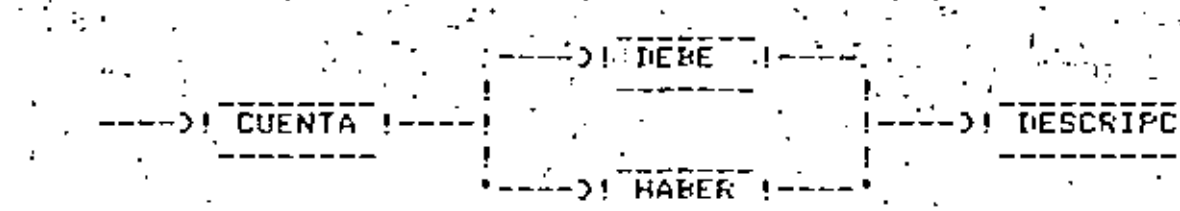
POLIZA:



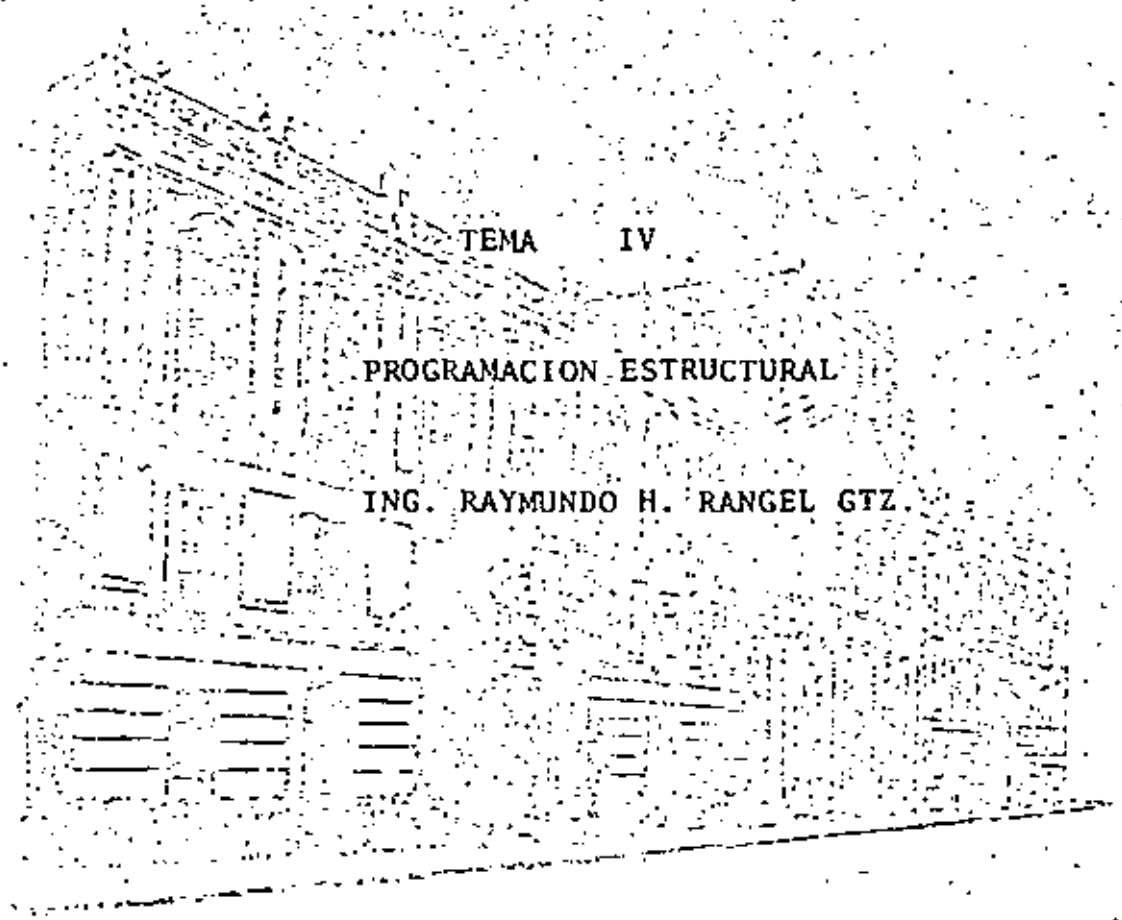
ENCABEZADO:



MOVIMIENTO:



METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION



TEMA IV

PROGRAMACION ESTRUCTURAL

ING. RAYMUNDO H. RANGEL GTZ.

MARZO, 1982

OBJETIVOS DE LA PROGRAMACION ESTRUCTURADA

Con frecuencia, se observa que los programas no

1.- satisfacen las necesidades del usuario
2.- no se producen a tiempo
3.- cuestan mas de lo estimado
4.- contienen errores y
5.- son difíciles de mantener

Esto ha motivado el desarrollo de nuevas técnicas y herramientas para reducir la complejidad en los programas.

L A M E T O D O L O G I A

La programación estructurada propone:

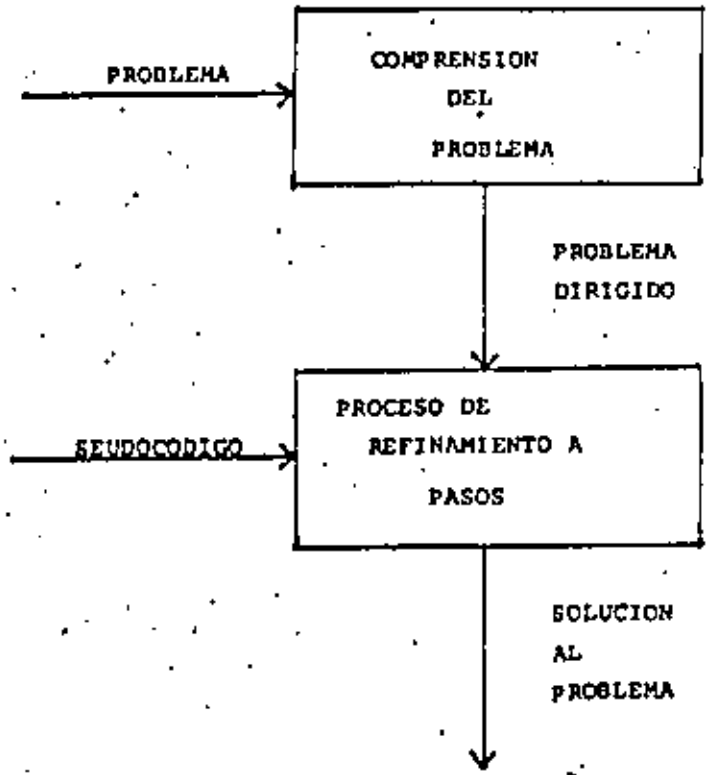
1.- Definir un conjunto pequeño de estructuras de control básicas para el desarrollo de código.

2.- Tener una firme comprensión del problema a fin de - - atacar los detalles del mismo.

Una vez que se hayan satisfecho 1 y 2, la programación estructurada sugiere atacar el problema en forma progresiva, yendo de lo general a lo particular. Este proceso se -- conoce como refinamiento a pasos.

El proceso de refinamiento a pasos se lleva a cabo mediante una secuencia de niveles de abstracción, procediendo -- de lo general a lo particular, es decir, se comienza planteando el problema en términos y entidades naturales al -- mismo y se finaliza al nivel de detalle del lenguaje de -- programación en que se implanta la solución.

El proceso de refinamiento y pasos es central a la programación estructurada.



E L S E U D O C O D I G O

El seudocódigo o seudolenguaje es un lenguaje intermedio - entre el lenguaje nativo del programador y el lenguaje de_ programación en que se intenta implantar la solución. El seudocódigo le permite al programador pensar en la lógica_ y expresar esa lógica en una forma semiformal sin tener -- que adentrarse en los detalles particulares de un lenguaje de programación.

Básicamente el seudocódigo difiere en 2 aspectos de un lenguaje de programación:

1.- No existen restricciones sintácticas para el uso del_ seudocódigo. Solo las estructuras de control básicas y el sangrado para mejorar las claridad del alcance de dichas estructuras, son las únicas convenciones_ aceptadas para el uso del seudocódigo.

2.- Cualquier operación se puede expresar a cualquier nivel de detalle, por ejemplo

```
incrementar   contador

ctdor  ←  ctdor  +  1
```

El seudocódigo permite al programador tratar el problema a diversos niveles de abstracción, ya que esta es la herramienta mediante la cual expresamos la solución de un problema durante el proceso de refinamiento a pasos, esto es, el seudocódigo es un lenguaje de diseño de programas (PDL).

El seudocódigo es una forma conveniente para documentar los estados de desarrollo del programa, lo cual permite a otros programadores revisar la función del programa antes de implementarlo en un lenguaje de programación, además de facilitar una valoración del estado de desarrollo en cualquier estado del proceso de diseño de programa.

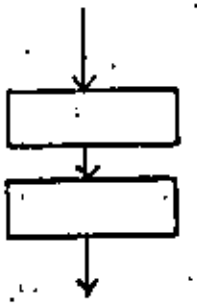
Debido a que el seudocódigo describe detalladamente el programa fuente completo, puede mantenerse como parte de la documentación del programa. Siendo así, el seudocódigo debe actualizarse cada vez que haya cambios en el programa fuente.

Existen pocas guías generales que hacen del seudocódigo efectivo como una herramienta de diseño de programas. Estas son:

1.- Haga del seudocódigo una extensión del proceso del pensamiento.
2.- Sangre el código para resaltar la estructura de la lógica.
3.- De nombres a los datos de tal manera que reflejen su intención.
4.- Mantenga la lógica simple
5.- Utilice las estructuras de ntrol básico permitidos el proyecto.

L A S E S T R U C T U R A S D E C O N T R O L B A S I C A S

1.- Secuencia

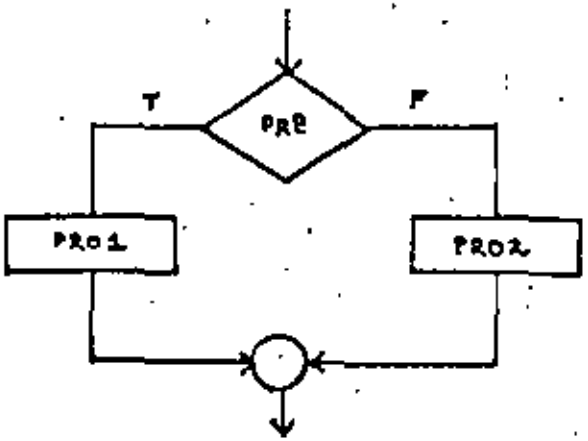


2.- Selección

seudocódigo.

```
IF  predicado   THEN
    proposición 1
ELSE
    Proposición 2
ENDIF
```

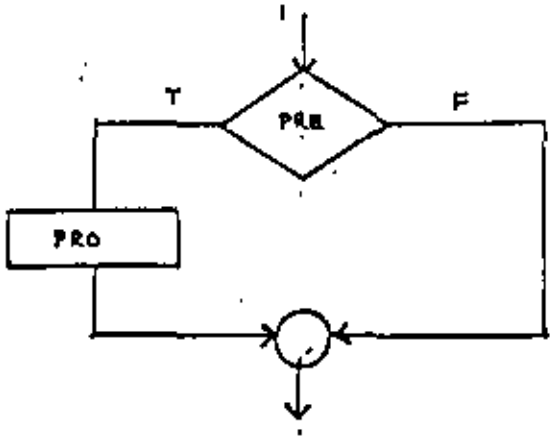
Diagrama de flujo



seudocódigo

```
IF  predicado  THEN
    proposición
ENDIF
```

Diagrama de flujo

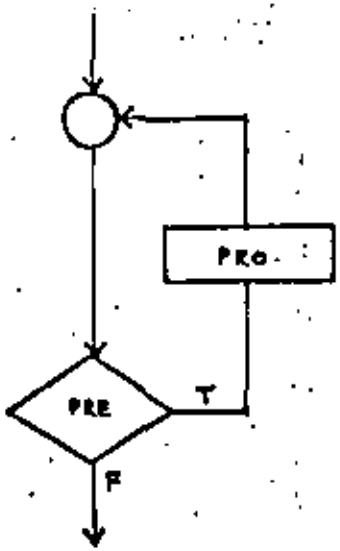


3.- Iteración

seudocódigo

```
WHILE predicado DO
      proposición
ENDWHILE
```

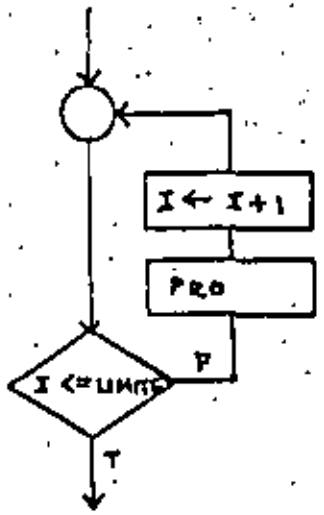
Diagrama de flujo



seudocódigo

```
FOR i ← expa 1  TO  expa 2  DO
    proposición
ENDFOR
```

diagrama de flujo



3.- Salidas

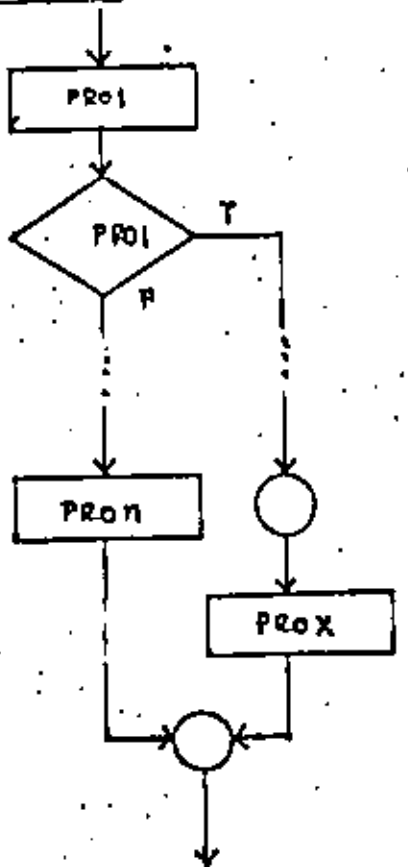
ESCAPE etiqueta
CYCLE etiqueta

seudocódigo

```
POSIT
      proposición 1
      QUIT POSIT IF (P1)

      proposición n
ELSE
      proposición x
ENDPOSIT
```

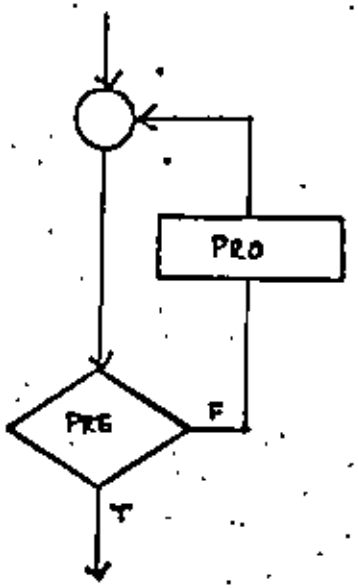
Diagrama de flujo



seudocódigo

DO
    proposición
UNTIL predicado

Diagrama de flujo



EXTENSIONES A LAS ESTRUCTURAS BASICAS DE CONTROL

1.- Selección

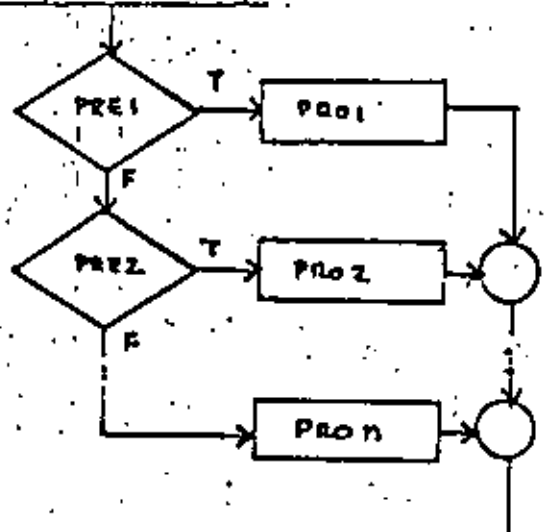
seudocódigo

IF predicado 1
    proposición 1
ORIF predicado 2
    proposición 2

.
.
.

ORIF predicado n-1
    proposición n-1
ELSE
    proposición n
ENDIF

Diagrama de flujo



EL PROCESO DE REFINAMIENTO A PASOS

Existen pocas guías para llevar a cabo este proceso:

Las siguientes guías deben considerarse en el proceso de - refinamiento a pasos

1.- Posponer detalles
2.- Tomar cuidadosamente desiciones
3.- Ser flexible
4.- Considerar los datos.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTES AL TEMA V

CONTROL DE CALIDAD

M. en C.. Marcial Portilla

ABRIL, 1982

PERSPECTIVES ON SOFTWARE ENGINEERING — MARVIN V. ZELKOWITZ

MAKING THE MOVE TO STRUCTURED PROGRAMMING — EDWARD YOURDON

AN EXAMPLE OF STRUCTURED DESIGN — BILL INMON

THE NEED FOR SOFTWARE ENGINEERING — WARE MYERS

SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS — DOUGLAS T. ROSS, JOHN GOODENOUGH, C.A. IRVINE

THE MYTHICAL MAN-MONTH — FREDERICK P. BROOKS, JR

WHY PROJECTS FAIL — STEPHEN P. KEIDER

Perspectives on Software Engineering

MARVIN V. ZELKOWITZ

Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, D.C. 20234, and Department of Computer Science, University of Maryland, College Park, Maryland 20742

Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliability and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed. Solutions from both the management and the programmer points of view are then given for many of these problem areas.

Keywords and Phrases: certification, chief programmer team, program correctness, program design language (PDL), software reliability, software development life cycle, software engineering, structured programming, top-down design, top-down development, validation, verification

CR Categories: 4.0, 4.6, 4.9

INTRODUCTION

Software development usually proceeds in one of two ways: either the programmer works alone in designing, implementing, and testing a software system, or he is a member of a group of from three up to several hundred, working together on a large software system. Although software engineering embraces both approaches, here we are interested mainly in large-scale program development.

When the Verrazano Narrows Bridge in New York City was started in 1959, officials estimated that it would cost $325 million and be completed by 1965. It is the largest suspension bridge ever built, yet it was completed in November 1964, on target and within budget [ENR61, ENR64]. No similar pattern has been observed when we build software systems larger than those which had been built previously.

Software is often delivered late. It is frequently unreliable and usually expensive to maintain. The IBM OS project, which involved over 5,000 man-years of effort, was years late [BROO75]. Why is bridge engineering so exact while software engineering flounders so?

Part of the answer lies in the greater ease with which a civil engineer can see the added complexity of a larger bridge than a software engineer the complexity of a larger program. Part of today's "software problem" stems from our attempt to extrapolate from personal experiences with smaller programs to large systems programming projects.

We begin here by outlining the general approach used in developing program products, emphasizing aspects which are still poorly understood. Later, we enumerate the techniques which have been used to solve these problems. We do not attempt to cover all of the relevant topics in depth, but we give many references for further reading.

Software engineers are currently study-

CONTENTS

ing the causes of these problems and the mechanisms of software development. They seek both constraints on programming which will render software less expensive and more reliable and also the theoretical foundations upon which programs are built. Software engineering is not the same as programming, although programming is an important component. It is not the study of compilers and operating systems, although compiler writers and operating system implementors use similar techniques. It is not electrical engineering, although electronics does provide the basis for implementing the computer [JEFF77].

Software engineering is interdisciplinary. It uses mathematics to analyze and certify algorithms, engineering to estimate costs and define tradeoffs, and management science to define requirements, assess risks, oversee personnel, and monitor progress.

1. STAGES OF SOFTWARE DEVELOPMENT

The complexity of a large software system surpasses the comprehension of any one individual. To better control the development of a project, software managers have identified six separate stages through which software projects pass; these stages are collectively called the *software development life cycle*:

- Requirements analysis;
- Specification;
- Design;
- Coding;
- Testing;
- Operation and maintenance.

Figure 1, a pie chart, shows the approximate amount of time each stage takes. The stages are discussed in the following subsections.

Requirements Analysis

This first stage, curiously absent from many projects, defines the requirements for an acceptable solution to the problem. The statement "Write a COBOL program of not more than 50,000 words to produce payroll checks" is not a requirement; it is the partial specification of a computer solution to the problem. The computer is merely a tool for solving the problem. The requirements analysis focuses on the interface between

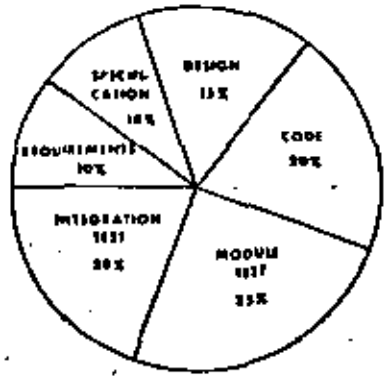


FIGURE 1. Effort required on various development activities (excluding maintenance).

the tool and the people who need to use it. For example, a company may consider several methods of paying its employees: 1) pay employees in cash; 2) use a computer to print payroll checks; 3) produce payroll checks manually; or 4) deposit payroll directly into employees' bank accounts.

Other aspects, such as processing time, costs, error probability, and chance of fraud or theft, must be considered among the basic requirements before an appropriate solution may be chosen. A requirements analysis can aid in understanding both the problem and the tradeoffs among conflicting constraints, thereby contributing to the best solution.

Hard requirements and the optional features must be distinguished. Are there time or space limitations? What facilities of the system are likely to change in the future? What facilities will be needed to maintain different versions of the system at different locations?

The resources needed to implement the system must be determined. How much money is available for the project? How much is actually needed? How many computers or computer services are affordable? What personnel are available? Can existing software be used? After the first questions are answered, project schedules must be planned. How will progress be controlled and monitored? What has been learned from previous efforts? What checkpoints will be inserted to measure this progress? Once all these questions have been answered, specification of a computer solution to the problem may begin.

Specification

While requirement analysis seeks to determine whether to use a computer, *specification* (also called *definition* [FIFE77]) seeks to define precisely what the computer is to do. What are the inputs and outputs? In the payroll example: Are employee records in a disk file? On tape? What is the format for each record in the file? What is the format for the output? Are checks to be printed? Is another tape to be written containing information for printing the checks offline? Will printed reports accompany the checks? What algorithms will be needed for computing deductions such as tax, unemployment and health insurance, or pension payments?

Since commercial systems process considerable amounts of data, the database is a central concern. What files are needed? How will they be formatted, accessed, updated, and deleted?

When the new system supersedes an older process (for example, when an automatic payroll system replaces a manual system), the conversion of the existing database to the new format must be part of the design. Conversion may require a special program which is discarded after its first and only use. Since the company may be using the older system in its day-to-day operation, bringing the new system online presents a problem. Can the old and the new systems run side by side for awhile?

The answers to these questions are set forth in the *functional specification*, a document describing the proposed computer solution. This document is important throughout the project. By defining the project, the specification gives both the purchaser and the developer a concrete description. The more precise the specifications are, the less likely will be errors, confusion, or recriminations later. The specifications enable test data to be developed early; this means that the performance of the system can be tested objectively, since the test data will not be influenced by implementation. Because it describes the scope of the solution, this document can be used for initial estimates of time, personnel, and other resources needed for the project.

These specifications define only what the system is to do, but not how to do it. Detailed algorithms for implementation are premature and may unduly constrain the designers.

Design

In the design stage, the algorithms called for in the specifications are developed, and the overall structure of the computer system takes shape. The system must be divided into small parts, each of which is the responsibility of an individual or a small

team. Each such module thus defined must have its constraints: its function, size, and speed.

As submodules are specified, they are represented in a tree diagram showing the nesting of the system's components. Figure 2 illustrates this for a typical compiler. This illustration, sometimes called a *baseline diagram*, is not by itself an adequate specification of the system.

Because the solution may not be known when the design stage starts, decomposition into small modules may be quite difficult. For older applications (such as compiler writing) this process may become standardized, but for new ones (such as defense systems or spacecraft control) it may be quite difficult.

A common problem is that the buyer of a system often does not know exactly what he wants, especially in state-of-the-art areas such as defense systems. As he sees the project evolve, the buyer often changes the specifications. If this occurs too often, the project may flounder. We discuss this problem later.

Coding

Coding is usually the easiest stage. High-level languages and structured programming simplify the task. In one study, Boehm [BOEH75] found that 64% of all errors occurred in design, but only 36% in coding. Hamilton and Zeldin [HAMI76] report that in the NASA Apollo project about 73% of all errors were design errors. We have mastered coding better than any other stage of software development.

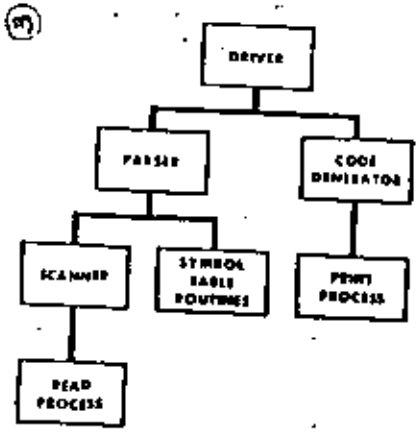


FIGURE 2. Sample baseline diagram for a compiler.

Testing

The testing stage may require up to half of the total effort. Inadequately planned testing often results in woefully late deliveries.

During testing the system is presented with data representative of that for the finished system; thus test data cannot be chosen at random. The test plan should, in fact, be designed early and most of the test data should be specified during the design stage of the project.

Testing is divided into three distinct operations.

1) *Module testing* subjects each module to the test data supplied by the programmer. A test driver simulates the software environment of the module by containing dummy routines to take the place of the actual subroutines that the tested module calls. Module testing is sometimes called *unit testing*. A module that passes these tests is released for integration testing.
2) *Integration testing* tests groups of components together. Eventually, this procedure produces a completely tested system. Integration testing frequently reveals errors missed in module tests. Correcting them may account for about a quarter of the total effort.
3) *Systems testing* involves the test of the completed system by an outside group. The independence of this group is important.

The buyer may also insist on his own systems test, or *acceptance test*, before formally accepting the product. Comparison of the performance of several systems (such as those of a given software product already available from several sources) is called *benchmark testing*.

During testing, many criteria are used to determine correct program execution. Among other important criteria, the program is considered correct if:

1) every statement has been executed at least once by the test data;
2) every path through the program has been executed at least once by the test data; and
3) for each specification of the program, test data demonstrate that the program performs the particular specification correctly.

These three different criteria show that there is no single acceptable criterion defining a "well-tested" program. Goodenough and Gerhart [GOOD76] proposed a set of consistent definitions for "testing" and showed that some of these definitions of testing are, in theory, insufficient. We return to this subject later. For a survey of good testing techniques, see [HUAN75].

Closely related to testing are verification and validation (V/V). A system is *validated* when testing shows that the system performs according to its specifications. A system is *verified* when it has been proved to meet its specifications. Current technology is inadequate for achieving both these objectives. A validated system may misbehave for cases not included in the test data. A verified system is correct relative only to the initial specifications and assumptions about the operating environment; formal proofs tend to be lengthy, making them subject to error or incredulity. *Certification* sometimes refers to the overall process of creating a correct program by validation and verification.

In certifying a program, three terms must be distinguished. A *failure* in a system is an event which marks a violation of the system's specifications. An *error* is an item of information which, when processed by the normal algorithms of the system, produces a failure. Since error recovery may be built into the program (for example, ON units in PL/I), not every error will produce a failure. A *fault* is a mechanical or algorithmic defect which generates an error (for example, a programming "bug") [DENN76a].

Reliability is a concept which must not be confused with correctness. A *correct* program is one that has been proved to meet its specifications. In contrast, a *reliable* program need not be correct, but gives acceptable answers even if the data or environment do not meet the assumptions made about them. We would like a system to be highly robust, that is, to accept a large class of input data and to process it correctly under adverse conditions. Parnas [PARN75] describes a correct system as one that is free from faults and has no errors in its internal data. A program is reliable if failures do not seriously impair its satisfactory operation.

Operating systems with "fail-soft" procedures illustrate the difference between reliability and correctness. A detected error causes the system to shut down without losing information, possibly restarting after error recovery. Such a system may not be correct because it is subject to errors, but it is reliable because of its consistent operation. A real-time program may be correct as long as a sensor reports correctly, but it may be unreliable if bad sensor readings have not been considered.

Operation and Maintenance

Figure 1 shows the disposition of software costs in developing a new project. But this can be the wrong chart! The activities noted in Figure 1 are only 25% to 33% of the effort required during the life of the system. Figure 3 illustrates that maintenance costs ultimately dwarf development costs.

No computer system is immutable. Since a buyer seldom knows what he wants, he seldom is satisfied. Probably, he will request changes in the delivered system. Errors missed in testing will later be discovered. Different installations will need special modifications for local conditions. The management of multiple copies of a system is another difficult problem that must be handled early in development. Once the first line of code is written, the structure of the resulting maintenance operation may already be fixed, so it is best to plan for it then.

The division of effort indicated in Figure 3 greatly affects system development. Because of hidden maintenance costs, techniques that rush development and provide

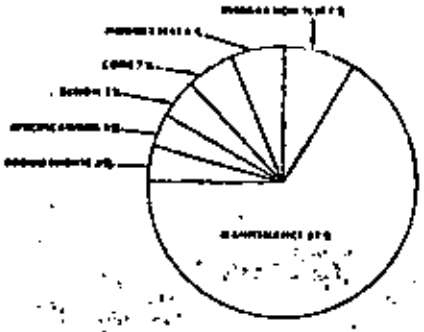


FIGURE 3. True effort on many large-scale software systems.

for very early initial implementation may be trading early execution for a much more extensive maintenance operation.

The maintenance problem is sometimes referred to as the "parts number explosion." For example, a certain system contains components A, B, and C. Installation I finds and reports an error. The developer fixes the error and sends a corrected module A' to all installations using the system.

Installations II and III ignore the replacement and continue with the original system. Installations I and II discover another error in module A. The developer must now determine whether both of these errors are the same, since different versions of module A are involved. The correction of this error involves correction of both A' (for I) and A (for II) yielding A" and A'". There are now three versions of the system.

To avoid this growth, systems often receive updates, called releases, at fixed intervals. A useful tool for dealing with myriad maintenance problems is a "systems database" started during the specifications stage. This database records the characteristics of the different installations. It includes the procedures for reporting, testing, and repairing errors before distributing the corrections.

Themes of Software Engineering

It should be clear that each software development stage may influence earlier stages. The writing of specifications gives feedback for evaluating resource requirements; the design often reveals flaws in these specifications; coding, testing, and operation reveal problems in design. The goals of software engineering are thus to:

- Use techniques that manage system complexity.
- Increase system reliability and correctness.
- Develop techniques to predict software costs more accurately.

In the following sections, we discuss approaches to some of these problems. The list of techniques is divided into management and programmer issues. Management issues concern the effective organization of personnel on a project. Programmer issues concern the techniques used by individual programmers to improve their performance.

2. MANAGEMENT ISSUES

A manager controls two major resources: personnel and computer equipment. This section surveys techniques for optimizing the use of these resources.

Size and Cost Control

A project may fail when management is not aware of developing problems; a year's delay comes "one day at a time" [BROO75]. Faced with catastrophic failure (for example, needed hardware is delayed six months), a resourceful manager can usually find alternatives. However, it is easy to ignore day-to-day problems (such as sick employees or many errors during testing).

Most problems occur at the interfaces of modules written by different programmers. Since the number of such interfaces is on the order of the square of the number of individuals involved, the problem becomes unwieldy when the number of persons in a development group grows to four or more.

As an example of the communications problem, assume that a single programmer is capable of writing a 5,000-line program in a year, and that a programming system requires about 50,000 lines of code and is to be completed in two years. Five programmers would seem to be sufficient (see Figure 4a).

However, the five programmers must communicate with one another. Such communication takes time and also causes some loss in productivity since finding misunderstood aspects will require additional testing. For this simple analysis, assume that each communication path "costs" a programmer 250 lines of code per year. Each of the five programmers, therefore, can produce only 4,000 lines per year and only 40,000 lines are completed within two years (see Figure 4b).

This means that eight programmers producing 3,250 lines per year are actually needed in order to produce the required 50,000. A manager is required for direction of this large effort. Therefore, in summary, eight programmers and a manager, each producing an average of 3,000 lines per year, are actually needed (see Figure 4c).

As we shall see, simply counting lines of code is not a good way to estimate productivity. The figures in this example are only given to illustrate a point, but they are representative of the problem. There are also techniques designed to limit this communications "explosion" and to increase programmer productivity.

Project Personnel

Software can usually be divided into three categories: 1) control programs (such as operating systems), 2) systems programs (such as compilers), and 3) applications programs (such as file management systems). A single programmer working on a control program can produce about 600 lines of code per year, whereas he can produce about 2,000 lines if working on a systems program and about 6,000 if working on an applications program [WOLV74]. The type of task certainly affects the productivity that can be expected from a given programmer. However, as the previous example demonstrates, the organization of personnel also affects performance. For example, with the approach of deadlines, docu-

FIGURE 4(a). Single projects: 5,000 lines per year = 50,000 lines for three years (no communication between programmers).

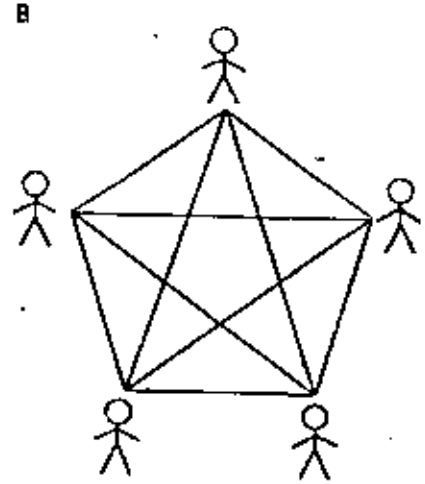


FIGURE 4(b). Five-member group: 4,000 lines per year = 40,000 lines in two years (10 communication pairs).

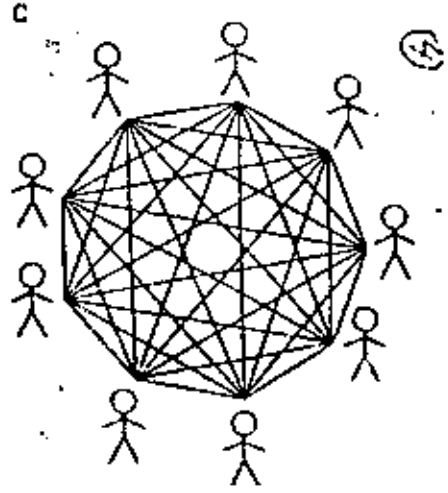


FIGURE 4(c). Nine-member team: 3,000 lines per year = 50,000 lines in two years (36 communication pairs).

mentation is often given lower priority. However, since 70% of the total system cost may occur during the maintenance state (where the documentation is heavily used), this may be a false economy of effort.

Use of a librarian is one way to avoid this problem. A librarian provides the interface between the programmer and the computer. Programs are coded and given to the librarian for insertion into the online project library. The actual debugging of the module is carried out by the programmer, but changes to the official module in the library are made by the librarian. The use of a library is further enhanced when an online data management system is used.

The use of a librarian has another beneficial effect. All changes in modules in the project library are handled by one individual and are easy to monitor; they are often reviewed by the project manager before insertion. This prevents "midnight patches" from being quickly incorporated into a system and forces the programmer to think carefully about each change. It also gives the manager disciplined product control and helps with audit trails.

On larger projects, a technical writer may perform much of the documentation, thus freeing programmers for the tasks for which they are most skilled.

The culmination of this trend is the *chief programmer team* concept developed by IBM [BAKE72]. The concept recognizes that programmers have different levels of competence; therefore, the most competent should do the major work, while others function in supporting roles. As the earlier example shows, interfacing problems greatly reduce programmer productivity. The chief programmer team is one way of limiting this complexity.

The chief programmer, an excellent programmer and a creative and well-disciplined individual, is the head of the team. He may be five or more times more productive than the lowest member of the team [BOEH77]. He functions as the technical manager of the project, designs the system, and writes the top-level interfaces for all major modules.

If a project is large, a team may also have an administrative manager to handle such responsibilities as budgeting time, vacations, office space, and other resources, and reporting to upper-level management. The administrative manager often administers several programming teams.

The backup programmer works with the chief programmer and fills in details assigned by the chief programmer. Should the chief programmer leave the project, the backup programmer would take over. This means that he also must be an excellent programmer. The backup programmer also fulfills an important role by providing the chief programmer with a peer with whom he can discuss the design.

There are also two or three junior programmers assigned to the team to write the low-level modules defined by the chief programmer. The term "junior" in this context means "less experienced," not "less capable." As Boehm states, the best results occur with fewer and better people.

Using the example illustrated by Figure 4, a chief programmer team of five individuals has only seven communications paths, and the chief programmer, being that rare individual, can produce more than his quota of 5,000 lines (see Figure 5). Thus productivity per programmer could be greater than 5,000 lines per year, instead of the previous figure of only 4,000.

The team has a librarian to manage the project library—both the online module library and the offline project documentation (also called the project notebook). The project notebook contains, among other things, records of compilations and test runs of all modules. It is important to the team structure, since all development is now accountable and open for inspection, and code is no longer the "private property" of any individual programmer.

Programmers have traditionally been reluctant to exhibit their products until completion, since discovered errors have traditionally been viewed as a personal failure. The absurdity of this approach is clear enough. If the ego element is removed from programming, programmers may openly ask others for advice when they need it, instead of trying to solve all problems themselves [WEIN71].

The team may include other supporting

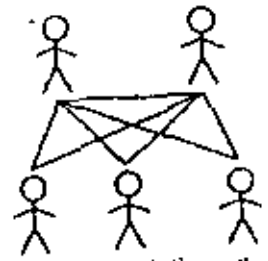


FIGURE 4. Fewer communications paths in a chief programmer team.

personnel such as secretaries and technical writers. Experience shows that ten is the upper bound to team size.

This structure, however, will not solve all problems in development. With a smaller number of individuals involved, competence is crucial. It is not possible to "work around" a nonproductive individual as one might do in a large project. There are also extremely large projects where a group of ten is simply too small to tackle development. Larger teams are not efficient.

A man-month, or the amount of work performed by one individual in one month, is a deceptive measure for estimating project productivity. A project requiring four programmers for a year cannot be completed by 48 programmers in one month. The example of the 60,000 line system needed in two years shows some of the problems inherent in trying to exchange programmers for time. "Adding manpower to a late software project makes it later" [BROO75]. New personnel divert existing personnel needed to train them; they require more supervision; they complicate communication and interfere with the design since they are unfamiliar with the project structure.

However, man-months do serve a purpose as a useful measure of project costs. By adding more data, such as the rate of using man-months, accurate cost estimation techniques can be utilized. These are explained in the following subsection.

Estimation Techniques

One of the most important aspects of engineering is estimating the resources needed to complete a project. As previously mentioned, the Verrazano Narrows Bridge in New York City was completed at the projected time and within the estimated budget. How was such accuracy achieved?

Most engineering disciplines have highly developed methods of estimating resource needs. One such technique is the following [GALL65]:

1) Develop an outline of the requirements from the Request for Quotation (RFQ);
2) Gather similar information, for example, data from similar projects;
3) Select the basic relevant data;
4) Develop estimates;
5) Make the final evaluation.

Although this approach has been advocated for software development, software projects have difficulty passing Step 1 [WOLV74]. Engineers have been building bridges for 6,000 years but software systems for only 30 years. Prior experience to develop the true requirements may not be available. Moreover, with very little background to build on, the developer has little knowledge of similar systems to use in evaluation (Step 2).

In developing the estimates (Step 4), the following tasks must be undertaken:

4a) Compare the project to similar previous projects.
4b) Divide the project into units and compare each unit with similar units;
4c) Schedule work and estimate resources by the month.
4d) Develop standards that can be applied to work.

Note that for Step 4a), the lack of previous experience presents a continuing problem. Also, for Step 4d), an adequate set of standards does not yet exist.

Experience is the key to accurate estimation. Even civil engineering projects may fail badly when established techniques are not followed. Although the Verrazano Narrows Bridge was the world's largest suspension bridge, its engineers had much experience with other similar structures. On the other hand, the Alaskan oil pipeline was estimated to cost $900 million, yet by mid-

1977 the cost had risen past $9 billion [ENR77]. In this case, the design was altered continuously as the federal government imposed new environmental standards (that is, changing specifications), and new technologies were needed to move large quantities of oil in a cold weather environment. Previous experience was only marginally helpful.

Results from computer hardware reliability theory are now starting to play a role in software estimation [PUTN77]. The cumulative expenditures over time for large-scale projects have been found to agree closely with the following equations:

$$E = K(1 - e^{-at^2})$$

where E is the total amount spent on the project up to time t, K is the total cost of the project, and a is a measure of the maximum expenditures for any one time period. This relationship is usually expressed in its differential form, called a Rayleigh curve:

$$E' = 2Kat e^{-at^2}$$

where E' is the rate of expenditures, or the amount spent on the project during year number t. Since 70% of the cost of a project occurs during the maintenance stage, it is not surprising that the maximum expenditures will occur just before the product is released, a time when it is usually assumed that the effort is winding down before termination (see Figure 6).

The Rayleigh curve has two parameters, K and a; however, a system can be described by three general characteristics: 1) total cost, 2) rate of expenditure, and 3) completion date. Two of these characteristics are enough to determine the constants K and a. When a project is initiated, the proposed budget is an estimate of K, and the available personnel permits a to be calculated. Assuming that requirement analysis determines that these figures represent an accurate assessment of the complexity of the problem, the estimated completion date (the date when the expenditures reach a maximum) can be computed, and thus cannot be set arbitrarily during the requirements or specification stage. This method provides the basis for a cost estimation strategy that has been applied to smaller projects in the 100 man-month range [BAIL78]. We may be close to a mathematical theory of cost estimation which will greatly reduce our need to "guess" at project costs.

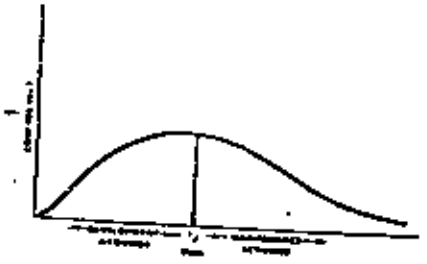


FIGURE 6. Yearly rate of expenditures approximates the Rayleigh curve. Total cost [area under curve = E; a = 1/(TA; rate = $2Kat e^{-at^2}$]

Milestones

A *milestone* is the specification of a demonstrable event in the development of a project. Milestones are scheduled by management to measure progress. "Coding is 90% complete" is not a milestone because the manager cannot know when 90% of the code is complete until the project itself is complete.

There are many candidates for milestones: publication of the functional specifications, writing of individual module designs, module compiling without errors, units that have been tested successfully, and so on. Milestones are scheduled fairly often to detect early slippage. PERT charts may be used to estimate the effects of slippage in one stage on later stages.

Reporting forms can give information useful for estimating when a future milestone will be reached. A general project summary, describing such overall characteristics as system size, cost, completion dates, or complexity, can be resubmitted with each milestone. Change reports can be submitted each time a module is altered. The use of a librarian probably means that such a form already exists. Weekly personnel and computer reports monitor expenditures. Although they add a minor overhead to the project, the information helps management keep abreast of progress [BAIL78, WALS77].

Development Tools

Compilers and certain debugging facilities have been available for some time. In contrast, other programming aids are new and experience with them is less extensive. Cross referencing, attribute listings, and symbolic storage maps are examples of such aids. Auditors or database systems can help to control the organization of the developing system. The Problem Statement Language/Problem Statement Analyzer (PSL/PSA) of the ISDOS project of the University of Michigan is one of the first database systems for providing a module library for storing source code, and includes a language for specifying interfaces in system design which can be checked automatically [TEIC77]. RSL/SSL is a similar system designed to specify requirements and to design interfaces via a data management system [DAVI77].

An alternative approach is the Programmer's Workbench developed by Bell Telephone Laboratories [DOLO76]. A PDP 11 based system provides a set of support routines for module development, library maintenance, documentation, and testing. Proper use of these facilities allows accessing information in an easier, controlled environment.

Reliability

Conceptual Integrity

Conceptual integrity, uniformity of style and simplicity of structure, are usually achieved by minimizing the number of individuals in the project. A chief programmer team greatly enhances conceptual integrity.

A small group minimizes contradictory aspects of a design. In the PL/I language, for example, the PICTURE attribute declaration may be abbreviated as either PIC or P, but in format specifications it may only be P [ANSI76]. In FORTRAN, the right side of an assignment statement can be an arbitrary arithmetic expression, but DO loop indices must be integer constants or variables, and subscripts to arrays are limited to seven basic forms [ANSI66]. These are difficult idiosyncracies to remember. They illustrate a lack of conceptual integrity that can arise when many people with different objectives become involved in a project. A consistent design is less prone to error because the user can follow a simple set of rules.

Continual System Validation

A *walkthrough* is a management review to discover errors in a system. In one study, TRW discovered that the cost of fixing an error at the coding stage is about twice that of fixing it at the design stage, and catching it in testing costs about ten times as much as it does in design [BOEH76].

A walkthrough is scheduled periodically for all personnel. In attendance are the project manager (chief programmer), the person reviewed, and several others knowledgeable about the project. One section of the system is selected for review and each individual is given information about that section (for example, design document for a design walkthrough, code for a coding walkthrough) before the review. The person being reviewed then describes the module under study.

The walkthrough is intended to detect errors, not to correct them. Also, the walkthrough is brief—not more than two hours. By explaining the design to others, the person reviewed is likely to discover vague specifications or missing conditions.

An important point for management is that the walkthrough is *not* for personnel evaluation. If the person reviewed perceives that he is being evaluated, he may attempt to cover up problems or present a rosy picture.

An informal yet very effective version of the walkthrough is *code reading*. A second programmer reviews the code for each module. This technique frequently turns up errors when the second reader, failing to understand some aspects of the code, asks the author for an explanation.

3. PROGRAMMER ISSUES

Each stage of the software development life cycle has its own set of problems and solutions. The most advanced techniques apply to the last stages; the first stages are the least developed. For example, testing and

debugging problems are apparent to every programmer, these tools are the oldest and most advanced. Techniques for improving coding were developed next. The most recent developments have related to requirements and specifications. Although many technical problems have not been solved, an effective methodology is emerging. Some of these techniques are presented in the following subsections.

Verification and Validation

Verification and validation (module and integration testing) of a system occupy about half of the development time of a project. Many debugging aids have been developed to facilitate this effort; most are implemented as programs to test some feature of a system.

Automated Tools

The earliest and most primitive debugging tools were the dump and the trace. A *dump* is a listing of the contents of the machine's memory. This listing can often reveal unintelligible data or errors. Unfortunately, a dump may not be taken until long "after the fact" and the cause of the error may not then be apparent. A *trace* is a printout showing the values of selected variables after each statement is executed. It may help a programmer to discover errors.

These techniques are not usually very effective because they supply much data with little or no interpretation. More advanced methods are needed to reduce this data to an intelligible form.

Flowgraph analyzers are capable of detecting references to variables which are never initialized or never reused after receiving a value; these usually indicate errors. Test data generators are also available. Assertion checkers validate that given conditions are true at indicated points of a program. Automatic verification systems have been implemented for small languages [KING69] and symbolic execution has been proposed as a practical means for validating programs in a more complex language. The ISL/PSA system is an example of a tool for assisting in design and specification. Symbolic dumps and traces are generated with compilers like PL/C [CONW71] or PUM [ZELK75]. Ramamoorthy and Ho [RAMA75] survey many of these tools.

Certification

Programs can be verified at several levels. Conway [CONW78] lists eight different verification conditions:

- A program contains no syntactic errors.
- A program contains no compilation errors or faults during program execution.
- There exist test data for which the program gives correct answers.
- For typical sets of test data, the program gives correct answers.
- For difficult sets of test data, the program gives correct answers.
- For all possible sets of data which are valid with respect to the problem specification, the program gives correct answers.
- For all possible sets of valid test data and all likely conditions of erroneous input, the program gives correct answers.
- For all possible input, the program gives correct answers.

Some people are optimistic that one day complete automatic program verification will be possible. Today's tools operate a posteriori, demonstrating that a given program works. Tomorrow's tools will also operate a priori, helping to develop programs which are correct before they are ever run. Such tools can reduce the amount of testing required for a completed project [DIJK76].

Verification techniques have the following general structures. A program is represented by a flowchart. Associated with each arc in the flowchart is a predicate, called an *assertion*. If A_i is the assertion associated with an arc entering statement S_i and A_j is the assertion on the arc following the statement, then the statement "If A_i is true, and if statement S_i is executed, then assertion A_j will be true" must be proved (see Figure 7).

This process can be repeated for each statement in a program. If A_1 is the assertion immediately preceding the input node to the flowchart (that is, the initial assertion), and if A_n is the assertion at the exit node (for example, the final assertion), then the statement "If A_1 is true, and the program is executed, then A_n is true" will be the theorem that states that the program meets its specifications (A_1 and A_n) (see Figure 8). This approach was formalized by Hoare [HOAR69] who defined a set of axioms for determining the effects upon the assertions (preconditions and postconditions) by each statement type in a language. Thus verifying program correctness reduces to proving a theorem of the predicate calculus.

Figure 7. Assertions A_i and A_j surround each statement of a program.

Certification technique development is still in a preliminary stage and does not meet the challenge of a modern large system. In addition, axiomatic certification is weak in the sense that the output assertion is proved true only if the program terminates. Axiomatic methods are incapable of proving termination. However, termination can often be proved informally by the programmer.

A typical approach to proving that program loops terminate is the following:

1) Find some number P that is always nonnegative within the loop.
2) Show that for each execution of the loop, P is decremented by at least a fixed amount.

If both conditions are always true, the loop must terminate before P becomes negative. A programmer who uses such rules, even informally, will seldom write nonterminating loops.

Consider this program fragment:

```
while x<y do
  ...
  x := x+1
  ...
end
```

Let quantity P be the expression $y-x$, and let $P(i)$ refer to the value of P during the ith execution of the loop. Because $x<y$ must be true for each next iteration, $y-x$ is always nonnegative and condition 1 is satisfied for each execution of the loop. Since the loop contains the statement $x := x+1$, $P(i+1) = P(i)-1$, satisfying condition 2. Therefore the loop must terminate.

Certification will not solve all our software problems, although it is an important tool. Gerhart and Yelowitz [GERH76] have shown that there are many published "certified" programs that contain errors. Even experts err.

Formal Testing

Goodenough and Gerhart [GOOD75] have clarified the concepts of testing. A *domain* is the set of permissible inputs to a program, and a *test* is a subset of the domain. A *testing criterion* specifies what is to be tested (for example, specifications, all statements, all paths).

A test is *complete* if the test meets all the requirements of the testing criterion, and a complete test is *successful* if the program gives correct results for each input in the test.

With these definitions, we can define program reliability and validity. A program is *reliable* if every found error is revealed by every complete test. A program is *valid* if every error is revealed by some complete test.

With these definitions, several important results can be proved. Among these are:

- If a program is both reliable and valid, then it is correct if and only if any complete test is also successful.
- The criterion "execute every path" is not valid; there exist programs all of whose test sets succeed, but which produce the wrong results for some input.

While this framework is somewhat technical and is not applicable to all programming, it is an important step in formalizing this area. We now have a basis for talking

Figure 8. Predicates A_1 and A_n specify input-output behavior of a program.

grammer's task is made easier when the computer does more work.

Structured Programming

A major development in facilitating the programming task is known as *structured programming*, which has been erroneously called "gotoless" programming. Fortunately, the debate about "to goto or not to goto" has mostly disappeared, and some clear ideas have emerged. The premise of structured programming is to use a small set of simple control and data structures with simple proof rules. A program then is built by nesting these statements inside each other. This method restricts the number of connections between program parts and thereby improves the comprehensibility and reliability of the program.

The if-then-else, while-do, and sequence statements are a commonly suggested set of control structures for this type of programming, however, there is nothing sacred about them. Knuth [KNU 74] has argued that the goto statement is irrelevant to the true goals of structured programming.

These simple control structures help programmers certify programs, even at an informal level. For example, a program can be represented as a function from its input data to its output data. Suppose $f(x)$ represents a segment of a program given by the following if-then-else statement:

$$\text{if } p(x) \text{ then } g(x) \text{ else } h(x)$$

Because functions g and h are simpler than function f, their specifications should be simpler. If their specifications are known, the overall function f is defined by

$$f(x) = (p(x) \rightarrow g(x)\ ,\ \neg p(x) \rightarrow h(x))$$

The programmer can express the formal definition of f in terms of the simpler definitions of g and h.

Languages such as ALGOL, PASCAL and certain subsets of PL/I contribute to good programming practices by providing these facilities. In order to repair FORTRAN's lack of structure, over 50 preprocessors for translating well-structured pseudo-FORTRAN programs into true FORTRAN have been developed [BOG 76]. An if-then-else has been added to the new FORTRAN-77 standard, although a general while is still missing from the language.

System Design

A technique related to structured programming is *top-down design*, in which a programmer first formulates a subroutine as a single statement, which is then expanded into one or two of the basic control structures mentioned earlier. At each level the function is expanded in increasingly greater detail until the resulting description becomes the actual source language program in some programming language.

Using this approach, also called *stepwise refinement* [WIRT 71, WIRT 74], the program is hierarchically structured and is described by successive refinements. Each refinement is interpreted by returning to other refinements of which it is a component. Concerning this method, Wirth states,

> I should like to stress that we should not be led to infer that actual programs come about by means of such well-organized, straightforward, "top-down" manner. Later refinement steps are often shown that earlier decisions are inappropriate and must be reconsidered. But the goal of *structured factorization* of a program seems admirable well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal arguments of correctness. The emphasis on clarity is particularly welcome if programs have to be adjusted to changed or extended specifications [WIRT 74, p. 251]

Operating systems are often modeled as hierarchies of *abstract* or *virtual* machines [BURS 77]. At the lowest level of the system is the physical hardware. Each new level provides additional *capabilities*, or allows the functions on data, and hides some of the details of a lower level. For example, if one level accesses the paging hardware of the computer and provides a large virtual memory for all other processes, other abstract machines at higher levels can be implemented as if they had unlimited memory since this detail is controlled by a lower level.

The concept of a *program design language* (PDL) is and in this development

about such concepts as reliability and correctness.

Mean Time Between Failure

While useful for focusing our attention, analogies with other engineering fields must be used with care. Reliability is one area of incomplete analogies. The concept of *mean time between failure (MTBF)* does not apply directly to software although it sometimes is used as if it does.

Systems built from physical components wear out; transistors fail; motors burn out; soldered joints break. This is also true for the hardware of the computer. However, the logical components of software are durable. A given program will always produce the same answer for the same input, as long as the hardware does not fail. When a software module "fails," it has been presented with an input that finally revealed an error present from the start.

The MTBF measures the time between revelations of errors. This, in turn, depends on the kinds of inputs presented. A compiler used only for short jobs from students may have a long MTBF; but if it is suddenly used for other applications, its MTBF may decrease sharply as unsuspected errors are exercised. A large MTBF can thus be interpreted only as an indication of possible reliability, not as a *proof* of it.

Error Days

Since formal certification of large classes of programs is still unattainable, techniques for estimating the validity of programs are still being considered. Most of these techniques measure the number of errors discovered, which are assumed to be representative of the total number of errors present in the system, and hence a measure of the reliability of the system.

Mills [Mill.76] defines an *error day* as a measure stating that one error remains undetected in a system for one day. The total number of error days in a system is computed by summing, for each error, the length of time that error was in the system. A high error day count may reveal many errors (poor design) or long-lived errors (poor development).

The assumption is made that if a program is delivered with a low error day count, then there is a good chance that it will remain low during future use. However, two major problems remain before this measure can be widely used. First, it is difficult to discover when a particular error first entered a system. Second, it may be difficult to obtain such information from the developer of a delivered product.

Programming Techniques

Several authors have mentioned that the number of lines of code produced by a programmer in a given time tends to be independent of the language used. This implies that higher level languages enhance productivity [Broo75, Hals77]. This is true even though assembly language programs are potentially more efficient, their potential is seldom realized in practice.

The goals in developing early higher level languages were to be able to express clearly an algorithm and translate it into efficient machine language programs. The efficiency of the resulting code was all important. This led to some anomalies in FORTRAN arising from the structure of the IBM 704 for which it was developed (for example, the three-way branch of the arithmetic IF). ALGOL, which was developed as a machine-independent way of expressing algorithms, contained concepts whose implementation on conventional hardware was inefficient (e.g., recursion, call-by-name); this may explain why ALGOL is not widely used.

By the late 1960s it was accepted that the language should facilitate writing the program and that the machine should be designed to create an efficient run-time environment. Today there is a definite shift toward using the language to make programming and documentation easier and to produce reliable and correct software.

This does not mean, however, that efficiency is ignored today. Whereas PL/I permits the writing of simple programs whose execution time is quite long, PASCAL was designed to exclude constructs whose machine code is inefficient. Since hardware is less expensive than programmers, reliability has become a major factor. The pro-

has been defined [CAIN75]. This type of language contains two structures: "outer" syntax of basic statement types, such as if-then-else, while, and sequence for connecting components, and an "inner" syntax that corresponds to the application being designed. The inner syntax is English statement oriented, and is expanded, step by step, until it expresses the algorithm in some programming language. Figure 9 represents an example of a PDL design.

It should be noted here that PSL/PSA and PDL complement each other. PSL/PSA is a specifications tool that validates correct data usage between two modules (interfaces). A system like PDL is useful for describing a given module at any level of detail. Both PSL/PSA and PDL can contribute to success in a large project.

Even though designed from the top down, many systems are implemented from the bottom up. Low-level routines are first coded with drivers to test them, then new modules, using these low-level routines, are added, and the system is built up.

Top-down development is another technique for implementing hierarchically structured programs. Here the top-level routines are written first and lower level routines, called *stubs*, are written to interface with these. The stubs return control after printing a simple message and may return some fixed sample test values. The stub is eventually replaced by the full module which now includes calls to other stubs. In this manner an entire system can be gradually developed.

If used carefully, this technique can be valuable, however, the system's correctness is assumed, not proved, until the last stub has been replaced [DENN76a]. The documentation specifies the assumptions on each stub. For example, if

$$f(x) = \text{if } p(x) \text{ then } g(x) \text{ else } h(x)$$

is a program fragment calling stubs g and h, then f will be correct only if the modules eventually replacing the stubs g and h are correct.

Via top-down development, a user sees the top-level interfaces in the system very early. He can then make changes relatively easily and soon. Another approach with the same goal is *iterative enhancement* [BASI 75]. Using this technique, a subset of the problem is first designed and implemented. This gives the user a running system early in the life cycle when changes are easier to make. This process is repeated to develop successively larger subsets until the final product is delivered.

Brooks [BROO75] believes that the first version of a system is always "thrown away," because the concrete specifications for a system are often not defined until the system is completed, a time when the initial product meets those specifications rather poorly. It is often cheaper and faster to rebuild a system from scratch than to try to modify an existing product to meet these specifications. However, a developer will often deliver such a modified system as a "pre-release" if a deadline is near and the purchaser is demanding results. The buyer then suffers with this version, replete with errors, until he throws it away or has the product rebuilt. Iterative enhancement can make rebuilding less chaotic since there is a running system (not meeting all the requirements) early in the development cycle.

```
max  PROCEDURE (lists,
     /* Find maximum element in a list */
     DECLARE (maximum, next) integer,
     DECLARE list list of integers,
     maximum = first element of list,
     DO WHILE (more elements in list),
       next = next element of list,
       maximum = largest of (next, old maximum),
       END
     RETURN (maximum);
     END max;
```

FIGURE 9. PDL of a program to find the largest element in a list (outer syntax is in upper case, inner syntax in lower case).

Performance Issues

The chosen algorithms and data structures have a much greater influence on program performance than code optimization or the programming language. Before choosing an algorithm, the programmer faces these questions:

- Can previously written software be used?
- If a new module must be written, what algorithms and data structures will give an efficient solution?

Programming languages usually include standard mathematical functions such as sine, logarithm, and square root. They give the programmer ready access to libraries of standard software packages. This allows the programmer to use results of previous work. In preparing programs for standard libraries, analysts have included many options in a single package. The effect can be a large cumbersome package which is inefficient because only a small part of it is applicable at any one time. This can be avoided by installing multiple versions of the module for each special case.

Many opportunities remain for more packaging and use of existing software. Difficulties in achieving this include:

- Identifying which standard algorithm to package. This is easier in mathematical areas such as statistical testing, integration, differentiation, and matrix computations than in many nonnumerical areas such as business applications.
- Transporting and interfacing with packaged software. Some progress has been made with programs stored in read-only memories which plug into microprocessors, or with interface processors on computer networks. A major problem area lies in interfacing software directly to other software, since there are no conventions. Some help is afforded by such concepts as the "pipeline" in UNIX, which provides a general communications channel between programs [RITC74].

Algorithm Analysis

Sometimes the program specification is not changeable, and the analyst must find the best possible algorithm. Sometimes, however, the specifications can be altered to permit a more efficient solution. In some instances we can show that there are no algorithms guaranteed to be efficient in all cases; here approximate algorithms that are efficient in most cases but need not give exact solutions must be used.

The fast Fourier transform illustrates the most efficient form for computing the Fourier transform, a technique useful in waveform analysis [COOL65]. This transform is based on a finite set of points rather than on a complex integral which is harder to compute. Language analysis (parsing) in a compiler illustrates how changing the specification can permit a more efficient solution. Any string of N symbols in an arbitrary context-free language can be parsed in time of order $O(N^{**}3)$ [YOUN67]; however, a programming language need not include all features of an arbitrary context-free language. PASCAL is an example of a language which can be parsed by a deterministic topdown parser in average time of order $O(N)$ [ALLO72]. If we are free to set language specifications, we can choose the language and be rewarded with efficient compilers.

Many practical problems, such as job scheduling or network commodity flow, involve enumeration of a combinatorially large number of alternatives and selection of a best solution. In these cases it may be better to restrict the search for a suboptimal but good answer. We recommend the paper by Weide [WEID77] for a discussion of the issues and a state-of-the-art survey of algorithm analysis.

Efficiency

In many cases the results of algorithmic analysis are not extensive enough to help the programmer; thus we need to offer techniques which can help locate and remove sources of inefficiency. One such tool is an optimizing compiler which, for some languages, can yield significant improvements [LOWR69]. The value of such tools, however, is limited [KNUT71] and may be realized only for programs which are used often enough to justify the investment in optimization.

One of the most powerful aids is the *frequency histogram*, which reveals how often each statement of a program is executed. It is not unusual to find that 10% of the statements account for 80% of the execution time [KNUT71]. A programmer who concentrates on these "bottlenecks" in his algorithms can realize significant performance improvements at a minimum investment. This technique has been used in some interactive operating systems, such as

UNIX and MULTICS, which started out as high-level language operating systems. Bottlenecks have been replaced by assembly language routines in less than 20% of the system.

Theory of Specifications

One area of software engineering that is now under study is system specifications. The objective is to state the specifications early using a metalanguage. This places restrictions on the design and may help establish whether the specifications are met.

An early example of such a specification was the so-called "gotoless programming" [DAHL68, KNUT74]. It is properly called "structured programming." It restricts the form of statements a programmer may use, but this restriction contributes to comprehensibility and enhances a correctness proof.

A second set of such rules employs the concepts of levels of abstraction, information hiding, and module interfacing to restrict access to the internal structure of data. Parnas [PARN72] formalized these ideas which were standard practices of expert programmers. He defines data as a collection of logical objects, each with a set of allowable states. Procedures can then be written to hide the representation of these objects inside separate modules. The user manipulates the objects by calling the special procedures.

Several languages that facilitate the use of these concepts have been developed. Among these are EUCLID [POPE77], CLU [LISK77], and ALPHARD [WULF76]. These languages permit programmers to define abstract data types having the property to encapsulate the representation of the logical objects [LISK75]. When concurrency is an issue, the use of abstract objects must be controlled by synchronization (for example, locks, signals); in this case the abstract type managers are called *monitors*.

Another kind of specification consists of "higher order software axioms" (HOS) [HANS76], which are a set of six axioms that specify allowable interactions among processes in a real-time system. One axiom prohibits a process from controlling its own execution, thereby ruling out recursion in a design. Another axiom states that no module controls its own input data space and is therefore unable to alter its input variables. While these axioms are not complete, they are a first step at formalizing specifications for system design.

SUMMARY

Boehm has stated seven principles that have helped organize the techniques discussed in this paper [BOEH76]

1) *Manage using a sequential life cycle plan.* This means to follow the software development life cycle outlined earlier. It allows for feedback which updates previous stages as the consequences of previous decisions become unknown. It encourages milestones to measure progress.

2) *Perform continuous validation.* Certify each new refinement of a module. Use walkthroughs and code reading. Display the hierarchical structure of the system clearly in all documentation.

3) *Maintain disciplined product control.* All output of a project—design documents, source code, user documentation, and so forth—should be formally approved. Changes to documents and program libraries must be strictly monitored and audited. Code reading, project reporting forms, librarians, a development library, and a project notebook all contribute to this goal.

4) *Use enhanced top-down structured programming.* PL/I and PASCAL have good control and data structures. Preprocessors exist which augment FORTRAN for these structures. Description techniques such as stepwise refinement, nested data abstractions, and data flow networks should be used.

5) *Maintain clear accountability.* Use milestones to measure progress, and a project notebook to monitor each individual's efforts.

6) *Use better and fewer people.* The chief programmer team, in which each individual is skilled and accountable for his actions, and good results are rewarded, aids in this effort.

7) *Maintain commitment to improve process.* Settle only for the best; strive for improvement. Be open to new develop-

ments in software engineering, but do not sacrifice reliability for modifiability while pursuing them.

Progress has been made in understanding how large-scale software systems are built, yet more needs to be done. Management aids must be improved and project control techniques developed. The role of software management is coming more to resemble that of engineering management in other disciplines. We can no longer afford costly mistakes when systems are so large and we depend so much on them. Most importantly, we must be patient; we need to gain experience on which future theories can rely.

ACKNOWLEDGMENTS

The author is indebted to Peter Denning for his detailed review and to the referees for their valuable comments on this paper. This work was partially supported by grant number DCR 74-11520 A01 from the National Science Foundation to the National Bureau of Standards.

REFERENCES

[AHU72] AHO A., AND ULLMAN J. *Theory of parsing, translation, and compiling.* Prentice Hall, Inc., Englewood Cliffs, N. J., 1972.

[ANSI66] *American Standard FORTRAN.* American Nat'l. Standards Inst., x3.9-1966, March, 1966.

[ANSI76] *American Standard PL/I.* American Nat'l. Standards Inst., x3.53-1976, Aug., 1976.

[BAKE72] BAKER F. T. "Chief programmer team management of production programming," *IBM Sys. J.* 11, 1 (1972), 56-73.

[BASI78] BASILI V., AND ZELKOWITZ M. "Analyzing medium scale software development," *Third Int. Conf. Software Engineering*, 1978.

[BASI75] BASILI V., AND TURNER A. J. "Iterative enhancement: a practical technique for software development," *IEEE Trans. Softw. Eng.* 1, 4 (Dec. 1975), 390-396.

[BOAU75] BOAUDE B., MCCLEAN R., AND VIENNOT D. "Some experience with automated aids to the design of large-scale reliable software," *Int. Conf. on Reliable Software*, 1975, ACM, New York, pp. 105-113.

[BOEH77] BOEHM, B. "Seven basic principles of software engineering," in *Infotech state of the art report on software engineering techniques*, 1977, Infotech International Ltd., Maidenhead, UK, 1976.

[BRIN77] BRINCH HANSEN P. *Architecture of concurrent programs.* Prentice Hall, Inc., Englewood Cliffs, N. J., 1977.

[BROO75] BROOKS F. P. *The mythical man-month.* Addison-Wesley Publ. Co., Reading, Mass., 1975.

[CAIN75] CAINE, S. H., AND GORDON, E. K. "PDL—a tool for software design," in *Proc. 1975 AFIPS Nat'l. Computer Conf.*, Vol. 44, AFIPS Press, Montvale, N. J., pp. 271-276.

[CONW78] CONWAY, R. *A primer on disciplined programming.* Winthrop Publishers, Cambridge, Mass., 1978.

[CONW73] CONWAY, R., AND WILCOX, T. "Design and implementation of a diagnostic compiler for PL/I," *Commun. ACM* 16, 3 (March 1973), 169-179.

[COOL65] COOLEY J. W., AND TUKEY J. W. "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.* 19, 90 (1965), 296-301.

[DAVI77] DAVIS C. G., AND VICK C. R. "The software development system," *IEEE Trans. Softw. Eng.* 3, 1 (Jan., 1977), 69-84.

[DENN76a] DENNING, P. J. "A hard look at structured programming," in *Infotech state of the art report on structured programming*, 1976, Infotech International Ltd., Maidenhead, UK, pp. 181-202.

[DENN76b] DENNING P. J. "Fault tolerant operating systems," *Comput. Surv.* 8, 4 (Dec. 1976), 359-389.

[DIJK68] DIJKSTRA E. "GOTO statement considered harmful," *Commun. ACM* 11, 3 (March 1968), 147-148.

[DIJK76] DIJKSTRA E. *A discipline of programming.* Prentice Hall, Inc., Englewood Cliffs, N. J., 1976.

[DOLO76] DOLOTTA T. A., AND MASHEY J. R. "An introduction to the programmer's workbench," in *Second Int. Conf. Software Engineering*, 1976, pp. 164-168.

[ENR161] "Everything about the Narrows Bridge is big, bigger, or biggest," *Eng. News Record* 166, June 29, 1961, 24-26.

[ENR164] "Narrows Bridge opens to traffic," *Eng. News Record* 173, Nov. 19, 1964, 33.

[ENR177] "Alaskan pipe coal probe hits snag," *Eng. News Record* 198, April 7, 1977, 14.

[FIPS77] FIPS, D. *Computer software management: a primer for project management and quality control.* Nat'l. Bureau of Standards, Inst. Computer Sciences and Technology, Special Publications, April 1977.

[GAFF83] GAFFNEY, P. F. *Project estimating by engineering methods.* Hayden Book Co., New York, 1983.

[GERH76] GERHART S., AND YELOWITZ L. "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Softw. Eng.* 2, 3 (Sept., 1976), 195-207.

[GOOD75] GOODENOUGH J. B., AND GERHART S. "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.* 1, 2 (June 1975), 156-173.

[HALS77] HALSTEAD M. *Elements of software science*, Elsevier North Holland, Inc., New York, 1977.

[HAMI76] HAMILTON M., AND ZELDIN S. "Higher order software—a methodology for defining software," *IEEE Trans. Softw. Eng.* 2, 1 (March 1976), 9-32.

[HOAR69] HOARE, C. A. R. "An axiomatic basis for computer programming," *Commun.*

ACM 12, 10 (Dec. 1969), 576–580, 583.

[HOWA75] HOWARD, J. C. "An approach to program testing," *Comput. Sust.* 7, 3 (Sept. 1975), 113–126.

[JEFF77] JEFFREY, S. R. AND LINDER, T. "Software engineering is engineering," in *IEEE Computer Science and Engineering Curricula Workshop*, 1977, IEEE, New York, pp. 112–115.

[KING69] KING, J. C. "A program verifier," PhD Dissertation, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa., 1969.

[KNUT71] KNUTH, D. "An empirical study of FORTRAN programs," *Softw. Pract. Exper.* 1, 2 (1971), 105–133.

[KNUT74] KNUTH, D. "Structured programming with statements," *Comput. Surv.* 6, 4 (Dec. 1974), 261–301.

[LISK75] LISKOV, B. AND ZILLES, S. "Specification techniques for data abstractions," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 9–19.

[LISK77] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. "Abstraction mechanisms in CLU," *Commun. ACM* 20, 8 (Aug. 1977), 564–576.

[LOWR69] LOWRY, E. S., AND MEDLOCK, C. W. "Object code optimization," *Commun. ACM* 12, 1 (Jan. 1969), 13–22.

[MILL76] MILLS, H. D. "Software development," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 265–273.

[PARN72] PARNAS, D. L. "On the criteria for decomposing systems into modules," *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058.

[PARN75] PARNAS, D. L. "The influence of software structure on reliability," in *Int. Conf. Reliable Software*, 1975, pp. 358–362, (ACM *SIGPLAN Notices* 10, 6 (June 1975)).

[PIKE77] PIKE, G. J., HOUSTON, J. J., LAWSON, H. W., MEDHURST, J. G., AND JACKSON, H. L. "Notes on the design of EUCLID," in *Proc. ACM Conf. Language Design for Reliable Software*, ACM, New York, 1977, pp. 11–18.

[POLI77] POLIAS, L., AND WOLVERTON, R. *Quantitative management: software cost estimating*, tutorial, IEEE Computer Society, Nov. 1977, IEEE, New York.

[RAMA75] RAMAMOORTHY, C. V., AND HO, S. F. "Testing large software with automated software evaluation systems," *IEEE Trans. Softw. Eng.* 1, 1 (1975), 46–58.

[REID76] REID, D. J. "The structured FORTRAN dilemma," *SIGPLAN Notices* 11, 2 (1976), 30–42.

[RITC74] RITCHIE, D. M., AND THOMPSON, K. "The UNIX time-sharing system," *Commun. ACM* 17, 7 (July 1974), 365–375.

[TEIC77] TEICHROW, D., AND HERSHEY, E. A. "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Softw. Eng.* 3, 1 (1977), 41–48.

[WALS77] WALSTON, C. E., AND FELIX, C. P. "A method of programming measurement and estimation," *IBM Syst. J.* 16, 1 (1977), 54–73.

[WEGB77] WEGBREIT, B. "A survey of analysis techniques for discrete algorithms," *Comput. Surv.* 9, 4 (Dec. 1977), 291–313.

[WEIN71] WEINBERG, G. M. *The psychology of computer programming*, Van Nostrand Reinhold, New York, 1971.

[WIRT71] WIRTH, N. "Program development by stepwise refinement," *Commun. ACM* 14, 4 (April 1971), 221–227.

[WIRT74] WIRTH, N. "On the composition of well-structured programs," *Comput. Surv.* 6, 4 (Dec. 1974), 247–259.

[WOLV74] WOLVERTON, R. W. "The cost of developing large scale software," *IEEE Trans. Comput.* 23, 6 (1974), 615–636.

[WULF76] WULF, W., LONDON, R., AND SHAW, M. "An introduction to the construction and verification of ALPHARD programs," *IEEE Trans. Softw. Eng.* 2, 4 (1976), 253–265.

[YOUN67] YOUNGER, D. "Recognition and parsing of context-free languages in time n^3," *Inf. Control* 10, 2 (1967), 189–208.

[ZELK75] ZELKOWITZ, M. V. "Third generation compiler design," in *ACM Natl. Comput. Conf.*, 1975, ACM, New York, pp. 253–258.

(REVISED MARCH 14, 1978; FINAL REVISION ACCEPTED MARCH 5, 1978)

Making The Move To Structured Programming

by Edward Yourdon

The best way to ensure that people will resist the change is to try to implement all the new techniques at one time.

The most common objection to structured programming takes the form, "Gosh, it sounds great and we'd like to do it, but. . . ." More specific attacks have been leveled against the PERFORM statement and other forms of subroutine calls, against nested IF statements, and against the elimination of GOTO statements. A more subtle and powerful form of attack is this: "Oh, you're just talking about modular programming; we've been doing that for ten years!"

For structured programming to have the proper opportunity to show its strengths and not be rejected outright by an organization, consideration should be given to these questions:

1. What will the specific objections be? Are there any potential disadvantages that may be experienced as a result of using the techniques?

2. Which of the techniques should be attempted first, assuming that you cannot use them all? For example, should you attempt to use structured programming first, and then try top-down design on a later project?

3. What kind of programming project should you begin with as an experiment to demonstrate the benefits of the structured programming techniques?

4. How should you evaluate the success of the experiment?

Common objections

It would be unrealistic to assume that structured programming, top-down design, chief programmer teams, structured walkthroughs (one programmer explains his code to others), program librarians, and structured design would be accepted without argument in any organization. Here are some of the more common objections:

1. Many managers and programmers point out that the structured programming techniques are primarily intended for new development projects; for maintenance of existing unstructured programs, they seem to be of limited use (though the librarian concept and the structured walkthrough concept would still be quite useful). This point is basically valid, though it is usually possible to add new

sections of code in a top-down structured manner, e.g., when completely new features are being added to a system at the request of a user.

This problem may be solved eventually with the aid of "structuring engines" that will automatically convert unstructured logic into structured form; while such an "engine" cannot magically transform "bad" code into "good" code, it will at least foster some standardization.

2. Some managers point out that their programming projects are typically too small to require a team of programmers; therefore, they argue, they don't need any of the new "programming productivity" techniques. Since most managers apparently are not prepared to fire most of their mediocre programmers and replace them with one highly competent chief programmer, we must accept this objection as a fairly valid one—but only for the chief programmer team concept.

There is no reason why the existing programmers in the organization, even working by themselves, could not use structured programming and top-down design.

3. Still other managers object to the cost of training their programmers in the techniques of structured programming. This training exercise is admittedly nontrivial, though it depends on the programmer's experience. (Junior programmers learn the techniques more easily than senior programmers; the author's group trained 120 programmers in an Australian government agency prior to beginning our payroll project, and of the 20 who scored at the top of the class, eight were novice programmers with less than six months experience.)

Ease of training also depends on the programming language; the techniques are generally easiest in PL/1, reasonably easy in COBOL, and more difficult in FORTRAN and assembly language. In general, we found that programmers require three to five days of classroom training to learn the techniques, and approximately one month of programming (when they are at least as productive as they were previously) to become comfortable with the new techniques. The investment in training is thus relatively small compared to the benefits that were discussed in the preceding section.

4. The manager often has to overcome technical objections raised by the programmers; these objections most frequently come from senior programmers, many of whom are now project leaders, who still fondly recall the "good old days" of the 1401 and the 650. The common programmer-oriented objections are: it is awkward and inconvenient at first; the programming language is inadequate for the strict discipline imposed by structured programming; it is not obvious that the new approach will actually reap the benefits discussed above; and finally, there is a concern that the top-down structured approach will lead to tremendously inefficient programs.

The awkwardness and inconvenience is largely a matter of training. The question of language adequacy can be a relevant one, and one answer might be to convince the programmers (and their managers!) to begin using PL/1 and the other ALGOL-like languages in preference to the primitive COBOL-like and FORTRAN-like languages.

The objection about efficiency can only be answered by appealing to the programmer's common sense: the battle for efficiency is generally won or lost at the system or program design level (e.g., by making sure the system is running on an efficient hardware configuration, and that it isn't doing things it wasn't intended to do), and not at the bit-fiddling level (i.e., where the programmer "wastes" a microsecond or two indulging in a subroutine call).

Obviously, there are some exceptions to this, in some real-time systems, for example; but as Professor Bill Wulf of Carnegie-Mellon Univ. points out, "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity."

5. In addition to programmer retraining costs, many managers object to the cost of developing new standards to conform to the new programming techniques. Some organizations may have only recently finished developing standards for testing—and they tend to be "bottom-up" standards, in direct contrast to the top-down methodology currently being advocated. While the cost of developing new standards may well be substantial, it is difficult to think of any way of avoiding it. Indeed, even if structured programming had not appeared on the scene, surely some new techniques would eventually appear, forcing the voluminous standards manuals to be rewritten . . . it seems to be an inevitable fact of life.

This may be an academic point, but I know of one large bank and one insurance company that have apparently rejected structured programming for this reason alone—all of which seems rather sad.

Some managers have expressed the fear that the structured programming concepts might not work (or that their programmers might not be sufficiently familiar with the new techniques) on a significant project whose failure would have disastrous consequences. There is a very simple solution to this problem: if you have not tried structured programming before, you probably should not use a critical project as a "guinea pig."

I have seen three projects in early 1975 fail in their attempt to use various aspects of structured programming. One midwestern company used top-down testing at the program level, but then integrated the programs in a bottom-up fashion to build a system—and they couldn't understand why the "top-down" approach had failed to give them miraculous results.

Another company in New England made an unsuccessful attempt at using the program librarian concept, but it appears that the librarian was a secretary who was required to spend six hours a day typing envelopes and the other two hours supporting the programming team.

Another programming project failed in its attempt to use the chief programmer concept. In apparent ignorance of the whole philosophy of the concept, their chief programmer did not write any code during the entire project!

7. Some managers are concerned that they may not be able to see the benefits of the new techniques. This is often because they have no statistics to compare their current techniques with the new ones. To be more blunt, many organizations have no idea how many lines of debugged code their programmers generate each day, nor how many test shots the average programmer requires before he delivers his program to the user. Nor have they any idea how many residual bugs are found in programs that have been delivered to the user. Thus, one of the first results of structured programming may be an unpleasant awareness of just how bad things are; while this may well be unpleasant, it can also be a healthy shock.

The lack of statistical evidence has been used as a reason for not using structured programming; it usually takes the form of: "Oh, well, I'm pretty sure our programmers are above average anyway, so we don't need to use these new techniques."

Sometimes the problem is more blatantly political: when management does find out how bad the programming productivity currently is, they try very hard to cover it up to avoid the obvious accusation that they have been doing their job poorly for the past several years. A battle of precisely this sort is going on in one of the larger dp organizations in Detroit, where one of the people on the research staff has made some rather interesting studies based on a sample of 100,000 PL/1 statements: the average module size was 900 statements (so much for small, independent modules!); only four statements were DO-WHILE statements (so much for the assumption that all PL/1 programmers instinctively know about the three basic forms of structured coding); in a substantial number of the programs, none of the IF statements had an ELSE clause; and various other statistics suggest that the programmers have been writing "rat's-nest" code for the past several years.

8. There is an interesting variation on the preceding objection: some managers worry that the structured programming techniques may improve the productivity of their programmers by only 10% instead of the five-fold improvement generally advertised. Of course, this may be because the programmers were already following an informal semi-structured, semi-top-down approach.

Nevertheless, some managers worry that they (and presumably their programmers as well) will be judged incompetent if they experience less than a five-fold improvement! One hardly knows what to say about this head-in-the-sand objection, except the obvious point that a 10% improvement in productivity (with commensurate improvements in software reliability and maintenance) is better than no improvement at all!

9. Finally, some managers suggest that the fanfare and publicity associated with structured programming may be a disguised form of the Hawthorne effect—i.e., the programmers are more productive because they know they are being observed. It is hard to believe that any manager who has the slightest familiarity with programming would believe this, though perhaps that is the problem—many dp managers are about as familiar with programming as they are with the theory of relativity.

Even if the Hawthorne effect were relevant, so what? Why fight it? On the Australian payroll project, I was criticized for giving the programmers tee-shirts that had the word "superprogrammer" emblazoned on the front. Some people felt this was an "unfair" method of attempting to increase their productivity! But if it contributed to a five-fold improvement in programming productivity, it was worth it!

Picking the right combination

As mentioned earlier, some organizations are concerned about the difficulty of implementing all of the new programming techniques simultaneously. In most such discussions, four major techniques are considered: structured programming, top-down implementation, chief programmer teams (with structured walkthroughs), and the program librarian. In even the most progressive organization, it can be difficult to implement all of these techniques at the same time; other organizations feel that some of the techniques are simply not applicable for their programming projects.

It is important to emphasize that while the four techniques are usually used together, they do not have to be. It is possible to use structured programming without top-down implementation, or top-down implementation without structured programming. Similarly, it is possible to use the chief programmer approach without using the librarian concept, or vice versa. And it is possible to use the chief programmer concept and the librarian concept without using structured programming or top-down design.

While each of the concepts can be used separately, some of them are almost inevitably joined with others. If an organization decides to use the chief programmer team approach, it almost always uses the librarian and some form of structured walkthroughs; on the other hand, I have seen several projects where the librarian concept was used without the chief program-

mer team concept. Similarly, if structured programming is used, top-down design is also used; conversely, if an organization decides to use top-down design and top-down testing, it is possible that they may elect not to use structured programming (this seems especially true in COBOL shops that have been brainwashed for years to avoid nested IF statements).

It is difficult to make any general suggestions about the order in which the techniques should be implemented in a typical organization. Perhaps the most important point to recognize is that a sharp distinction can be made between the technical concepts of structured programming, top-down implementation, and structured design, and between the organizational concepts of chief programmer teams, structured walkthroughs, and program librarians. One should choose the combination of techniques that will give the greatest improvement for the least effort.

The librarian concept can usually be implemented fairly easily, since it does not affect the user and does not require any major retraining on the part of the programmers (though it does get them to change their attitudes: "Whaddya mean I can't type my own program to the time-sharing terminal? So what if I can only type two words a minute?"). On the other hand, it does require some standards to be developed for proper use of the librarian, as well as some training for the librarian; and it does cause some problems for organizations that find they have no budgets, no "job titles," and no place in the organization chart for the librarian. Except in government agencies and some other large bureaucratic organizations I have visited recently, these are usually minor problems.

Top-down design, structured programming, and the chief programmer concept tend to have a larger impact on an organization, and should be implemented with more care. Structured programming generally implies "GOTO-less" programming (or at least less GOTO programming), which is a jolt to many experienced programmers. The chief programmer team approach usually includes the concept of structured walkthroughs which suggests that each programmer should make a formal presentation of his code to all of the programmers on the team for their review and criticism. This too is a jolt to the average programmer (as a programmer at Shell Oil in London said, "Reading someone else's program is like reading their personal mail—it's an invasion of privacy in which civilized people simply do not indulge").

Top-down design and top-down testing are a bit easier for the programmer to swallow, but they can cause severe management disruptions in some cases. They imply, for example, that a computer is available for testing at an early stage in the project (in contrast to the common management policy of not supplying machine time until the final stages of the project).

None of these problems is insurmountable; indeed, many organizations have implemented all of the techniques simultaneously without experiencing any major catastrophes. Nevertheless, the cautious manager may wish to begin with only one new idea at a time: specific conditions within the organization will usually dictate which technique is to be implemented first.

A good pilot project

In some organizations, the techniques of structured programming and top-down design, after some experience with them, are considered "intuitively obvious" to managers and programmers alike. Once exposed to the concepts, everyone begins using the techniques without any significant prodding. Many organizations start using structured programming with a great deal of trepidation: in most cases, this is done by using the techniques on an "experimental" project.

This leads to an obvious question: what kind of project should be used as an experiment? Experience with several organizations during the past two years suggests the following:

1. *The project should be visible.* If the experiment is an academic project that nobody will use, then nobody will care about the results. The project should be a "real" one—one that users will use, and one whose results people will care about. Indeed, this is one of the reasons the New York Times project had such a profound impact upon IBM and the rest of the industry—it was real.

2. *The project should be nontrivial.* One of the keystones of the "structured" approach is its attempt to break complex tasks into simpler ones. By working with less complex program components, the programmer is less likely to make mistakes, and he is more likely to produce something that can be maintained. However, if the program is already quite trivial (e.g., less than 100 lines of code), the improvement in productivity and maintenance will not be as obvious. The experimental project should be at least 2 man-months in duration.

3. *The project should be noncritical.* There are enough problems in becoming familiar with structured programming and top-down implementation in a relatively small project; there is no point in putting additional pressure on the programmers by forcing them to use the techniques on a critical project—one whose failure will have disastrous consequences in the organization.

On the other hand, if management and programmers agree that the techniques are "intuitively obvious" and feel relatively comfortable with the techniques (a more and more common occurrence, considering that most programmers now graduating with a B.S. in Computer Science have had a healthy exposure to structured programming), then there should be no danger.

There is the case of an organization that was faced with an "impossible" project and in desperation used it as a pilot structured programming project. The Australian Taxation Dept. (roughly equivalent to the IRS) was urged in 1973 to adopt structured programming, top-down design, chief programmer teams, and program librarians to assist in a project involving conversion of machines, conversion of languages, and redesign of major existing systems within a timeframe that would otherwise have been considered impossible. The results were highly successful.

Evaluating the experiment

Clearly, the object of an experiment is to gain information for future reference: a structured programming experiment should give the organization valuable information about the usefulness of the techniques for future projects. From the discussion in the previous sections, we see that there are several statistics the project manager should try to capture; most of these can be gathered by answering the following questions:

a. Was the project finished on schedule? How accurate were the estimates for milestones during the project? At various stages during the project, was it possible to estimate accurately the amount of coding remaining to be done?

b. How productive were the programmers? In particular, how many debugged statements per day were they able to produce? Also, how did the programmers distribute their working time during the project (some tentative results from an experiment at Aetna Life & Casualty suggest that slightly under 6% of the programmer's time was spent on code reviews and walkthroughs, which seems to squelch the common complaint that these things take too much time).

c. How efficient were the resulting programs? This may be difficult to judge unless the program is a redesigned version of an earlier (presum-

ably unstructured) program. However, the manager and/or the programmers should be able to make some qualitative judgments about the presence or absence of substantial overhead in the final program.

d. How much test time was required for the project? Specifically, was the test time distributed fairly evenly throughout the project?

e. What was the ratio of development costs to maintenance costs? Is it significantly better or worse than other "unstructured" projects within the organization?

f. How effective were the structured walkthroughs? Roughly how many bugs were found per man-hour of walkthrough, and roughly how long would it have taken the original programmer to find those bugs? More important, how many of the bugs would have remained unnoticed until the program began running in production?

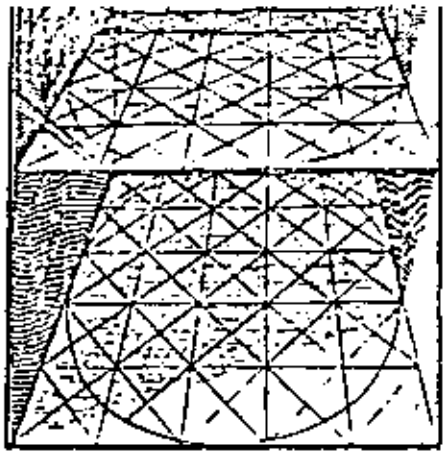
I find that a reviewing audience can easily spot five to six bugs in a 200 statement program within 15 minutes. It is important to recognize that the programmer who wrote the code was usually convinced that his code was correct, and his test data would usually be a self-serving attempt to confirm that feeling.

g. How many bugs were discovered during user acceptance testing? How many bugs were discovered after several months of production? How does this compare with other "normal" programs in the organization? □

Mr. Yourdon, president of Yourdon Inc., has consulted and lectured on program design and online computer systems in the U.S., Canada, Europe, and Australia. He began his career at Digital Equipment Corp., where he developed assemblers, FORTRAN IV Executives, and math libraries for various machines. At General Electric, he developed an operating system for a hospital information system on the GE-435. He has authored several books and numerous articles, and is currently completing a two-volume series, "Advanced Programming Techniques."

DATAMATION
reprint

An Example of Structured Design



An Example of Structured Design

by Bill Inmon

Producing 11 lines of code per hour through structured design and programming is one of the advantages.

Quick and accurate development of computer programs is a longstanding goal of the data processing community. Recently programmers and program designers have begun to formalize some of their practices by using concepts of structured programming and structured design. How do these concepts interact with the goals of quick and accurate program development?

An example of how one system was developed using these tools in a limited amount of time is given here. The specifics of the system may not be important—it happened to be a cost accounting system, necessitated by a large government contract, which broke down the cost of the final product to costs of component parts at the lowest level. However the problems that arose in the development of this system, and the way they were handled, are important because these same basic problems usually occur in development of most types of systems.

System overview

Here is a profile of the system under discussion:

1. Length of system development —4 months.
2. Manpower—14 man-months (7 programmers—2 full time, 5 part time).
3. Scope of project—tasks included internal design (i.e., the building of detailed programming specifications from a broad description of the problem), programming, unit testing, integrated testing, documentation, and the building and maintenance of a large data base.
4. Total number of lines of coding —approximately 31,000, not including system-support coding, such as Job Control coding, test-data generators, etc.
5. Language—COBOL interfacing the data base with DL/I.
6. Machine—IBM 370/145, with TSO.
7. Data base environment.
8. Programmer coding productivity @ 200 hours per month—that is, approximately 11 lines *per hour*.
9. Number of programs and modules—50
10. Median module size — 5.4 pages.
11. Type of application—financial.
12. Technical complexity—mild.
13. Programming logic complexity —moderate.
14. Design considerations—structured programming, structured design.

A group of programmers was organized into a "chief programmer team." The chief programmer was responsible for the design of all programs. He did the actual coding of some critical programs, and organized and coordinated the programming of the other programs in the system. Two programmers were on the team full time and five other programmers contributed to the team effort. Programs were designed adhering to the principles of structured design. Principally, programs were kept small and designed "functionally." Large programs were divided into modules. The coding of programs was structured. No GOTO statements were allowed, and other conventions of structured programming were followed.

The overriding constraint on the system development was time. Because of contractual obligations program design, programming, debugging, and implementation had to be completed in four months. The deadline was not extendable.

The structuring of the data base reflected a strong user orientation and was not designed for easy program manipulation. At the outset of the project, only two programmers had extensive data base experience. In preparation for future on-line considerations, the system was to be updated by transactions. Reports were generated directly from the data base. In terms of programming logic complexity, the system was straightforward except for several internal relationships. Technically, the complexity of the system was not beyond the ability of the chief programmer team.

Design goals

The design goals of the system were greatly influenced by these program development priorities:

1. Speed of development
2. Program and algorithmic accuracy
3. Maintainability of programs
4. System flexibility

STRUCTURED DESIGN

The traditional consideration of production run efficiency was only of incidental importance. It was not ignored; it simply gave way to the higher priorities. The major attention was directed towards the building of a workable system. Once the requirements of the system were met, consideration was given to run time efficiency (i.e., when there was enough time to make such a consideration, which usually was not the case).

Several schemes were contemplated for organizing the project. The only way deemed feasible because of the time constraint was to write program specifications while concurrently programming and testing other parts of the system. Obvious pitfalls to this approach were recognized before proceeding; however, because of the lack of time, no other method was possible. This type of project organization probably should be used only when the situation demands it. In such a case, very close control and coordination is an absolute must because poor communications can lead to a large scale waste of effort.

To attain the design goals, all large programs were highly modularized. As much as possible, a module was designed for general usage, so that it would be reusable in other parts of the system. In this way repetitious coding was eliminated or at least drastically reduced. Each module was physically separate from any other module, i.e., the source text which comprised a module was developed, compiled, and stored apart from any other module. The intent of the design was to make modules small, about four pages of source text per module. However, the size of a module was ultimately determined by its function. Each module had a limited and well-defined function, and the environment of the module was likewise limited and well-defined. In this manner, the overall functions of a program were broken into smaller, isolated functions.

Early in the system design, it was recognized that there were different levels of functionality along which a program could be divided. An example of levels of functionality is shown by the difference between two large updating programs that were part of the system. Both programs were to read transactions and update the data base according to the contents of the transaction.

Another way to view an updating program is from the traditional actions of adding, deleting, or replacing data in the data base. In this case, as a transaction enters the program, it is categorized into one of the three major functions, and is handled along those functional lines.

Program flexibility and modularization

The flexibility of the programs in the system was difficult to assess. However two characteristics of the system point to the fact that structured design produces some flexibility. One is that changes in it were made with a minimum of effort. The largest change required four days, and the next largest, less than a day. The second characteristic is the fact that errors were quickly located and corrected. It appeared that the major factor contributing to system flexibility was the isolation by function that was achieved by structured design. When a problem occurs, it is easy to eliminate many modules from consideration since their functions may have nothing to do with the problem.

Data coupling was extensively used in the interfacing of modules with each other. (Data coupling occurs when all input and output to and from the called module are passed as parameters or arguments—i.e., as data elements). The independence gained here enhanced both the reusability of a module and the isolation of a module by function from other modules.

In other ways, the modular approach accelerated the development of the system. The physical separation of modules meant that more than one person could simultaneously contribute to a program. In fact at one time, five programmers were actively working on modules of the same program.

A byproduct of using a modular approach was the lessening of the total learning time involved in the translation of programming specifications into code. This occurred because, as a programmer completed a module, he was assigned another similar one. He still had the previous coding fresh in mind and could therefore quickly grasp the new programming specifications.

Another feature of adopting a modular approach occurred when a programmer completed a module and was then free to work on other systems. In a conventional system, a programmer is tied to a program until he finishes it; and if a problem in his area of expertise arises while writing the program, something must suffer. However, if a program is broken into small modules, many breaking points result naturally, and the problems involved in conventional methods of programming are minimized.

Despite advantages in using a modular approach, there were also some disadvantages. One problem arose in the interfacing of modules. In spite of careful attention given to the flow of data to and from other modules, there still were some errors. The number and order of parameters, their characteristics, and the interpretation of their values were in some cases misunderstood. Another problem arose in the organization necessary for the direction of several people working on the same problem simultaneously. Modules were being developed so rapidly that coordination of testing, linking, and integrating them became a large task.

Structured walkthrough

One of the techniques that led to the quick completion of the project was that of the structured walkthrough. A walkthrough, or mental execution of the program by the programming team, was done for each program.

After a programmer had compiled a program and scrutinized it for obvious errors, he sent the chief programmer and several other programmers involved copies of his source text. After time was allowed for the group to examine the text (usually a few hours), the team met and collectively performed a mental execution of the program. A list of errors was made as each logical path was followed. The interface with modules either calling or called by the module being examined was carefully checked as well. At the end of the walkthough, the list of errors was given to the author.

In this manner most errors were caught before any testing had occurred. Also, the team members who reviewed the text became familiar with a part of the system other than their own. This helped to establish a common base of understanding of all components of the system.

Coincidentally, while the team reviewed the interface with other modules, errors (usually from miscommunication) were also spotted in other modules. It was also not uncommon to have a program or module execute correctly the first time it was tested. One of the major factors in the success of the system was the shortness of the time spent in testing, and the major contributor to it was structured walkthroughs.

Program testing

One reason why testing and debugging went smoothly was the fact that the design of the system included programming specifications for debugging. Not only did programming specifications define the function of a program or module, but they also required variables to be inserted and manipulated solely for the purpose of debugging.

The most effective technique was using an activity variable in each

module. Initially the activity variable is set up as inactive. As a module was invoked, the value of the activity variable was changed to indicate that it was active. As control was relinquished, the variable was returned to the inactive state. When a dump occurred, it was easy to trace the path of active modules, arriving at the final

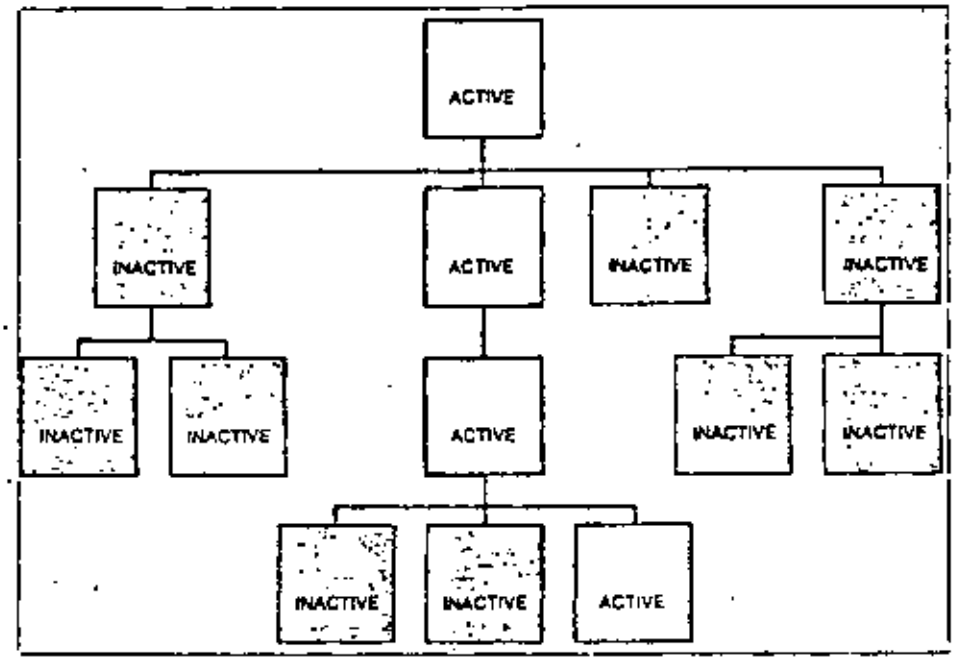


Fig. 1. By using an "activity" variable in each module, the flow of control through the program can be reconstructed and non-active modules ignored in debugging.

one in which activity had occurred (see Fig. 1).

Another useful technique was to display the parameters of a module at the time of invocation. Because of data coupling, the complete set of relevant data was available for determining the state of the module.

An additional technique that proved to be useful was the gathering of statistics internally by each module. Typically, a module would keep track of how many times it had been invoked, the parameters it had been called with, what type of internal results it had generated, etc. Data of this type made reconstruction and analysis of program activity an easy task. They also helped to identify areas of code that had never been tested. Also, it helped to locate areas of code that were critical in the efficient running of the program.

Summary

The most pleasing and surprising aspect of the project was the smoothness with which debugging and testing were accomplished. This smoothness is reflected by the high programmer productivity rate of approximately 11 lines per hour. We felt that several factors contributed to this success—structured walkthroughs, the breakdown of large programs into modules, and structured programming.

The major criterion, that of speed of development, was certainly enhanced by structured design. A modular programming approach alone would not have made it possible to write the large programs needed in the amount of time allotted.

The other design goals—flexibility and maintainability—cannot yet be effectively evaluated since the system has remained relatively stable in its short lifetime. The few changes made do point to a high degree of maintainability. Structured design, used in conjunction with the complementary techniques of structured programming and walkthroughs, did make possible a level of speed and accuracy in system development not previously attainable. Based upon our experience with developing this system, structured design is as powerful in practice as has been predicted in theory.

Mr. Inmon is a chief programmer for GTE Sylvania Electronics Systems Group in Mountain View, Calif. He received an M.S. in computer science from New Mexico State Univ.

30

THE NEED FOR SOFTWARE ENGINEERING

Ware Myers
Contributing Editor

Introduction

Early in this decade a set of programming practices began to appear that seemed to offer a way out of the software difficulties accompanying the development of large systems. These practices, developed by Brooks,[1] Baker,[2] Dijkstra,[3] Mills,[4] and others, included structured programming, top-down development, chief programmer teams, HIPO (hierarchy/input-process-output) documentation, development support library, and structured walk-throughs. But despite the increasing amount of software development and its rising cost relative to the defense budget, corporation expenditures, and even the gross national product, the new programming techniques have not been adopted by acclamation. McClure,[5] surveying the scene at COMPCON '76 Spring, saw "the great masses of programmers conducting their business exactly as they did five years ago." Nor was there the slightest sign in McClure's 5-year projection of "strong winds of change." His intuition was later supported by a survey of major Los Angeles area corporations,[6] which concluded that, for all the fanfare, "the techniques are simply not widely used."

Why are modern programming practices propagating so slowly? In the judgment of some seasoned observers, the reason lies in the complexity of the techniques and the difficulties management and programmers face in implementing them. As with modern management practices, modern programming practices are intangibles that have to be disseminated by "soft" means such as education and training.

Fortunately, there are positive forces at work—the Department of Defense, NASA, IBM, defense and space contractors, software houses, and universities. The professional societies cover the area in their conferences, tutorials, and journals. The General Accounting Office is studying the use and value of software development techniques. These influences, together with the growing realization that projected software development costs are becoming a greater factor than hardware costs in deciding to develop a system, are literally forcing progress to be made.

The software predicament

The general character of the software predicament can be seen clearly, although consistent numbers with which to characterize it more precisely are hard to come by. Because less expensive hardware is bringing more applications within economic reach, the amount of software to be developed is increasing. Also, because more software is already in existence, there is more of it to be maintained. But the productivity of programmers is improving rather slowly, especially by the standards of hardware price/performance, with the result that the overall cost of software development is tending to increase. Or, if this increase is being limited by budgetary considerations, opportunities to apply computer technology more fully to both old and new applications are being passed over.

Software growth. Estimates of the overall cost of software development and maintenance in the United States range from $15 to $25 billion.[7] At DOD it is running in the $3 billion-per-year range. It represents 4-5 percent of the Air Force budget, 6 percent of the NASA budget.

According to Walter R. Beam,[8] deputy for advanced technology to the assistant secretary of the Air Force for research, development, and logistics, the number of functions—most of them new—being done by software is growing at a prodigious rate. Beam, who gave the software technical keynote at

COMPCON 77 Fall in Washington, D.C., last September, estimated this rate to be a factor of 2 every three years. The rate is probably greater in defense than in industry, because DOD is moving rapidly from analog to digital weapon systems. For example, recently purchased aircraft are heavily software controlled because these new systems are more effective than the old hardware systems.

In another area Gnostic Concepts* projects data processing spending, exclusive of office automation, to increase at about 15 percent per year, while information systems business grows at better than 20 percent.

Operations type systems for steel mills, retail stores, banks, etc., are growing at about twice the rate of other applications, according to Joe M. Henson, vice president for market planning at IBM's Data Processing Division, in another technical keynote address given at COMPCON 77 Fall.[3] Because these systems are inherently complex, they require more programming manhours than simpler systems. Henson projected the number of on line systems to grow from 9500 in 1975 to 23,000 by 1980. More systems are coming in the area of management and technical planning, such as forecasting, modeling, and design, and they will demand more data. Meeting this need through large integrated data bases will again add to the volume of software development.

Last November the computer industry was reminded that not only do business system manufacturers not agree on standardized programs; users aren't settling for them anyway. "We're going the other way as customers demand even more diversity," Jay R. Hosler, computer systems consultant, told Interface West.[4] Small business systems are only a part of the total software picture, but they exemplify one of the ways the programming workload grows.

Maintenance ratio. Because a great deal of software is already in existence, some of it of low quality, more and more effort, of necessity, has to be devoted to maintenance. Henson, for example, estimated that up to 80 percent of his company's application development resources are devoted to maintenance. Similar figures were reported by Elshoff,[5] who noted that 75 percent of General Motors' commercial software effort was spent on maintenance. This ratio, according to Elshoff, is fairly typical of large-industry software activities. As more software manpower goes into maintenance, less is available for new development.

Productivity. Estimates of the long-term productivity improvement rate of programmers range from about 3 percent to 7 percent. For example, IFIP President Richard I. Tanaka estimated the current rate to be about 3 percent per year.[6] Henson gave a 4 percent to 7 percent rate, depending on the assumptions made, for the 30 year period from 1955 to 1985.

Within IBM, on large programming projects, "productivity, measured in terms of the number of lines of code produced per programmer, has improved at about 20 percent over the years, due in part to the increasing use of higher level languages," according to Ted Climis of IBM, addressing the keynote session of COMPSAC 77.[7] Climis, a vice president of his company's General Products Division, expected this rate to continue, but noted that productivity data on programs of under 20,000 lines of code is more variable and seems to depend largely upon the individuals involved.

Indeed, as Climis' last statement suggests, measuring programmer productivity is a complicated undertaking. For example, one investigator, finding that reported productivities ranged from one line of code per hour to 30 or more, concluded that more precise definitions of a program, a manday, and even a line of code itself were needed. Using his own definitions Johnson[8] found a productivity range on 16 commercial systems-programming products ranging from about 9 lines of code per hour average of five smaller projects to about 3 lines of code per hour average of 11 larger projects. In general, his results were close to those published earlier by Fred Brooks.[9]

Software-hardware ratio. Because the cost of hardware is dropping rapidly—an order-of-magnitude improvement in the hardware price/performance ratio every 10 years—while software productivity improves only slowly, the cost of software relative to hardware is increasing. This change has been noted by many observers. Tanaka, for example, saw information processing becoming the most labor-intensive of all industries by 1985, if better methods were not used. Brain characterized it as a "cottage industry"—hardly an image compatible with the notion of computers as typifying modern technology.

In NASA the software/hardware ratio was about 2:1 five years ago. Writing for *Datamation* in 1973,[9] Boehm projected the ratio for the Air Force as going to 10:1 by 1985. On one existing program, the World Wide Military Command and Control System, the ratio was already in that vicinity—$722 million for software to $50-$100 million for hardware. This system, with 35 locations, ties the President and the Pentagon with commands in the United States, Europe, and the Pacific, using airborne command posts in part.[10]

Reliability. The enormous size of WWMCCS offers a dramatic insight into another dimension of the software predicament: the need for correct operation. This need was highlighted by a recent news report by Greg Rushford, who writes on national security subjects: "The record of military communications is replete with failures, stemming in no small part from the system's complexities."[11] He attributed to communications difficulties at least part of the blame for such serious incidents as the

Gulf of Tonkin crisis in the Vietnam war, the bombing of a U.S. warship off the Sinai peninsula by the Israelis, the Pueblo affair off North Korea, and others. These breakdowns in communications occurred before the new computer-controlled system was installed, but they underscore the critical importance of its correct operation.

Before modern programming came into use, programs were large, complex, neither modular nor portable, almost unreadable, and expensive to maintain.

Program quality. What were typical programs like before modern programming practices? Elshoff[4] analyzed 120 PL/I programs collected from General Motors commercial computing installations in late 1973. He concluded that the individual programs were not modularized and were quite large—853 PL/I statements on the average. They were very complex, not portable, almost unreadable by others, and expensive to maintain.* From interviews with GM programmers who had worked elsewhere, he learned that this state of affairs seemed to be typical of other installations as well, especially Cobol installations. Perhaps this was roughly the stage of the programming art when modern programming practices appeared on the scene.

Need for discipline

The notion that a more disciplined approach to programming would alleviate the software problem has been current for at least 10 years. It was, in fact, the belief that systematic engineering methods could be applied to the software process that led to the coining of the term "software engineering" in 1968.[5]

The engineering way. The very term, engineering, implies "that the entire development of a product from initial conception through testing and maintenance is organized in an orderly, manageable way. The quality, performance, and cost of the product must be predictable, and an appropriate compromise between cost and reliability must be achieved."[6]

Engineering development generally proceeds through a series of stages: product planning, specification, design, documentation, fabrication, and test, with engineering change control running through the sequence. The development of any particular product may pass through this sequence several times, as model prototype and finally production. Engineers are trained in these stages from school onward.

A series of "dragons" enforce the disciplines: the machinist or technician wants an exact print he can build to, the drafting manager insists that documentation follow the rules, component engineers try to standardize the building blocks, and manufacturing engineers pour over the drawings to establish that they are complete.

The software way. The software process differs from the hardware process in many ways. For one thing the fabrication step, in the sense of reproducing the program tapes, is insignificant. Also, the habit of going through the stages several times has not caught on in spite of Brooks' advice: "Plan to throw one away; you will, anyhow."

In addition, past practice has tended to start with instruction writing and then work back to the earlier stages of design and requirements. This method worked well enough on the relatively small programs that characterized the early history of programming, but it ran out of steam on large developments.

Another factor in the programming way is the backgrounds of its practitioners. They tend to come from more varied sources than do the hardware engineers, who are usually the products of a systematic 4- or 5-year curriculum. Programmers may have degrees in mathematics, English, history, journalism, or whatever. Whatever they have learned—and it may be a great deal—it probably does not include engineering methods. This lack of common background may be part of the cause for Brooks' complaint that "techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering."

A comparison. It seems intuitively that systematic development procedures would lead to better results. Daly[7] attempted to measure the difference in the results obtained by the disciplined engineering approach and the less disciplined programming approach before modern programming practices. For this purpose he studied the recorded statistics of a large real-time system consisting of 160,000 instructions and 170,000 logic gates (not counting gates in duplicated circuits). He felt the size and complexity in each area were about the same.

Here is what he found:

- twice as much effort had been required to develop an instruction as a logic gate;
- four times as many design maintenance corrections had been made per instruction as per gate;
- four times as much cost had been incurred for design maintenance of software as hardware.

Daly thought he recognized four underlying reasons:

- management techniques and development procedures were more advanced in hardware than in software;
- hardware designers were more experienced and employed a more "structured design."

*Elshoff warned that the conditions were as of 1973. Since then GM has made many improvements.

- the basic building blocks used in software design (i.e., different types of source statements) were more numerous and complex than the AND, OR, and NOT concepts used in hardware;
- hardware got a double dose of testing and evaluation, one by itself and the second as a natural byproduct of software testing.

Software engineering process

In recent years the stages of the software process have been analyzed by many researchers and practitioners.[4] This work is summarized in Table 1, which presents an ordering of the eight main stages of software development and a second-order listing of some of the management and design practices characteristic of each stage.

The concept of a design stage in software and its separation from the programming stage is relatively new. In the design stage the intent is to work out completely and unambiguously the software system necessary to meet the specifications, using English, pidgin English, or a program design language. No instructions or code are written in this stage.

Managers complain that programmers resist the new ideas, while programmers retort that managers don't understand the problem.

The purpose is to create a logical structure which can be checked back against the specifications and internally within its own structure before proceeding to the additional labor of writing instructions in a high- or low-level programming language.

There may be exceptions, however, to the write-no-instructions rule. One is when some novel problem must be solved at the programming level to assure that the design level structure is workable. Another is reiteration, in which problems occurring in a later stage have to be fed back and necessarily cause changes in an earlier stage.

Table 1. The software development process

1 REQUIREMENTS ANALYSIS AND DEFINITION[21]

2 SPECIFICATIONS [22] AS A
- CRITERION FOR AGREEMENT BY USER AND DEVELOPER ON THE FUNCTIONS REQUIRED
- REFERENCE POINT FOR DESIGN DEVELOPMENT
- COMPARISON POINT FOR PROGRAM VERIFICATION
- PLANNING POINT FOR PROGRAM TESTING

3 DESIGN (IN THIS STAGE MANY OF THE NEW PROGRAMMING PRACTICES ARE USEFUL)
- TEAM CONCEPT CHIEF PROGRAMMER SUPPORT LIBRARIAN ETC
- TOP DOWN DESIGN OR STRUCTURED DESIGN COMPOSITE DESIGN STEPWISE REFINEMENT HIERARCHICAL OR MODULAR DECOMPOSITION
- PROGRAM DESIGN LANGUAGES OR PSEUDO CODE OR PROGRAM ENGLISH (THESE ARE NOT PROGRAMMING LANGUAGES)
- PRODUCT WORKBOOK OR UNIT DEVELOPMENT FOLDER
- FORMAL REVIEW PEER REVIEW ETC
- DOCUMENTATION—E G HIPO

(MANY OF THESE PRACTICES CARRY THROUGH TO OR HAVE THEIR COUNTERPART IN THE NEXT STAGE)

4 PROGRAMMING
- VARIOUS LEVELS OF PROGRAMMING LANGUAGES
- STRUCTURED PROGRAMMING FOR CODING
- INTERNAL OR SELF DOCUMENTATION
- STRUCTURED WALK THROUGH OR CODE REVIEW
- DEBUGGING

5 VERIFICATION AND TESTING
- CORRECTNESS (TO SPECIFICATIONS)
- RELIABILITY (IN USER ENVIRONMENT)
- TOP DOWN TESTING
- AUTOMATED AIDS

6 PERFORMANCE [2]
- EFFICIENCY TIME MEMORY SIZE
- QUALITY
- ADAPTABILITY (FLEXIBILITY MAINTAINABILITY)

7 OPERATION AND MAINTENANCE
- DESIGN OR PROGRAMMING ERROR CORRECTION
- MINOR UPDATING OR ENHANCEMENT

8 CONFIGURATION MANAGEMENT
- BASELINING AT VARIOUS STAGES AS AN AID FOR FEEDBACK CHANGES
- PROBLEM REPORTING
- CHANGE CONTROL BOARD (OR EQUIVALENT BODIES)

Modern programming practices. Although these practices have been widely discussed in the literature, Bolton[5] had to drop one-third of the companies he started out to survey because they had not even considered using them. At panel discussions, managers complain from the podium that programmers resist the new ideas, while programmers retort from the floor that managers are too far from the nitty gritty to understand the problem. One can only conclude that some people need further information about modern programming practices.

Table 2 defines the core practices briefly (the main purpose of this article, after all, is to establish the need for and the value of these techniques, not to elucidate them in detail). For that purpose the practices are referenced. In addition, Freeman and Wasserman have annotated 10 full-length books on various aspects of software design in their tutorial.[6] They also reprinted 21 of the more useful papers.

Several books are too new to be listed by Freeman and Wasserman. Thayer's[23] volume, produced at the Jet Propulsion Laboratory under a NASA contract, covers the development stages of Table 1 and the programming practices of Table 2 (and others) in a systematic manner. Hoyles and Mellichamp[24] cover a somewhat narrower field—top-down development, structured programming, stepwise refinement, and structured walk throughs—but

also treat structured programming in three common languages: Cobol, Fortran, and PL/I.

The DOD view

The Department of Defense has taken steps to encourage more effective software engineering. For example, a 1976 directive provided guidelines intended to create a discipline of software engineering.[1] Earlier, in March 1974, the Army Computer Systems Command and the Air Force Rome Air Development Center jointly sponsored a contract with IBM Federal Systems Division to document everything that was known about structured programming technology. This effort resulted in a 15-volume series.[2]

However, it is difficult for DOD to dictate just how contractors will go through the design and development process. It does not attempt to do so for hardware. Its proper sphere is to establish endproduct requirements.

Fortunately, a number of major software contractors have responded to the challenge of modern programming practices and have created disciplined development processes, taking advantage of the new practices in various ways. Not many people in the defense community disagree on the general value of those processes today.

Problems remain. There is still the problem of getting contractors to progress from where they are to the software engineering system that each one thinks is best or, perhaps, to the system he

Table 2. Some definitions of modern programming practices.

CHIEF PROGRAMMER TEAMS[3,4]

To reduce the coordination problems on large projects, Mills suggested the use of teams, each headed by an especially capable chief programmer who would be assisted by specialists in each function needed to support him. Brooks characterized it as the "surgical team." Coordination would be the province of the chief programmer alone, thus reducing the number of points involved in project communication by a factor of five or six.

DEVELOPMENT SUPPORT LIBRARIAN

One of the team members is the librarian. He is responsible for the programming product library containing both machine- and human-readable material. This function helps to transform programming "from a private art to public practice."

TOP-DOWN DEVELOPMENT[5]

Once requirements are learned up, the development process decomposes the proposed system into a series of levels in a hierarchy, beginning at the top and working down. The highest level is then designed, coded, and subsequently tested, but using shells with dummy code to stand in for lower level units that are involved, and so on.

MODULAR DECOMPOSITION[6-8,11]

To isolate the entire system into independent partitions, each module is constructed to work with others on control signals and data transfers, but to be uninvolved in the detailed internal structure of other modules. With strict module interfaces carefully specified, the relatively independent modules become easier to code, test, and later change than more dependent modules.

STRUCTURED DESIGN[12]

Structured design is a set of techniques for reducing the complexity of large new programs by dividing them into independent modules. Working with separate pieces permits the programmer to code, debug, test, and modify a functional module with minimal effect on other modules of the entire system. Concentrating effort in this way enhances efficiency and quality and reduces bugs. Moreover, to the extent that the independent modules are portable, further systems can be developed with less need for new code.

PROGRAM DESIGN LANGUAGE[13-15]

Intended to be comparable to the blueprint in hardware, program design languages strive to communicate the concept of the software design in all necessary detail, using a limited or structured version of English, sometimes called pidgin English or pseudo code.

PROJECT WORKBOOK[1]

Design efforts inevitably produce much written material—memorandums, explanations, reports. The idea is to capture and organize it so as to be sure it reaches all who need to know and is available for use later.

HIPO—HIERARCHY/INPUT PROCESS OUTPUT[6,16]

This part documentation, part analytical technique consists of hierarchy charts and the corresponding input process output charts. The hierarchy chart is a set of boxes, similar to an organization chart, showing each function and its division into subfunctions. For each function or subfunction, an input process output chart, roughly similar to the block diagram in logic design, shows the inputs and outputs and the processes joining them. If the HIPO charts themselves are arranged in a hierarchy, the techniques can be used to graphically document top-down design or structured design.

STRUCTURED PROGRAMMING[6,10]

To enhance readability and maintainability, a program is structured so that the logic flow proceeds from beginning to end without arbitrary branching. This approach is based on the theorem that any program with one entry and one exit can be constructed from only three control structures: SEQUENTIAL, IF-THEN-ELSE, and DO-WHILE. It is analogous to the formation of complex logic functions from only AND and NOT building blocks.

STRUCTURED WALK-THROUGH[10]

The structured walk-through, sometimes called peer review, is the old design review with sophisticated modifications.

- The reviewers take the initiative to plan and run the session; management does not attend, thus encouraging a non-defensive atmosphere.
- The new design and programming practices provide an understandable structure the reviewers can readily walk through.[17]
- The emphasis in the session is on error detection. The reviewers are solely responsible for finding errors; the author does later.

thinks the government would like him to use. And assuming modern programming practices as a group are effective, there is the further problem of determining just which practices are most suitable to particular applications. And inasmuch as modern programming practices cost budget money, there is the problem of finding out which ones are most cost effective. There is the problem—is software engineering sufficiently mature to be ready for standardization? Many observers think there is still much to learn and organizations should continue for some time yet to experiment with new techniques.

Software analysis. One clear need is for more data on the software development process. To meet this need, the Air Force is setting up through the Rome Air Development Center a software analysis center.[4] It is to gather program development statistics in order to determine the factors that influence the productivity of programmers and the cost of software development and maintenance. A second task is to understand the nature, causes, and effects of software failures. To accomplish these purposes, it appears that some definitions will have to be hammered out to make data from various organizations comparable.

Correlations. One way to establish the value of modern programming practices is to do correlation studies. Such studies attempt to correlate one or more programming practices against various measures of the effectiveness of the software development process, such as cost, errors, and programmer productivity. It would be interesting to know if some of the practices are more effective than others. In an attempt to answer this kind of question, the Rome Air Development Center commissioned a number of studies, three of which were reported to COMPCON Fall 77.

Cost relation. The first study, by Rachel Black of Boeing, measured the effects of modern programming practices on software development costs for five projects of Boeing Computer Services.[5] However, two of the projects were very small, so in view of the belief that productivity on small projects is highly dependent on the individuals involved, the results from these two projects have been eliminated from the data summarized here.

The key comparison was made between man-months for the projects as forecasted by Boeing's traditional estimating procedure and man-months actually incurred. The traditional procedures, said to predict costs within ± 15 percent, had not assumed the use of modern programming practices. However, the projects, as executed, did employ a variety of new practices.

The general effect of using modern programming practices was positive, as shown in Figure 1. The average improvement of the three projects, weighted for size, was 73 percent. This figure represents the difference between the forecast man-months and the actual man-months, divided by the forecast man-months.

Boeing found modern programming practices reduced actual costs over forecast costs by 73 percent.

The practices employed on the three projects are listed in Table 3 in the order of Black's estimate of their impact on cost. It does appear that some practices are more effective than others on the cost dimension.

These three projects, as well as the two smaller projects previously discarded, also utilized top-down design techniques, structured programming, and programming support tools and libraries. Because the two smaller projects had shown little improvement using these techniques, Black had eliminated them as contributors to the benefits the three larger projects achieved—probably unwisely. However, her interviews with project personnel

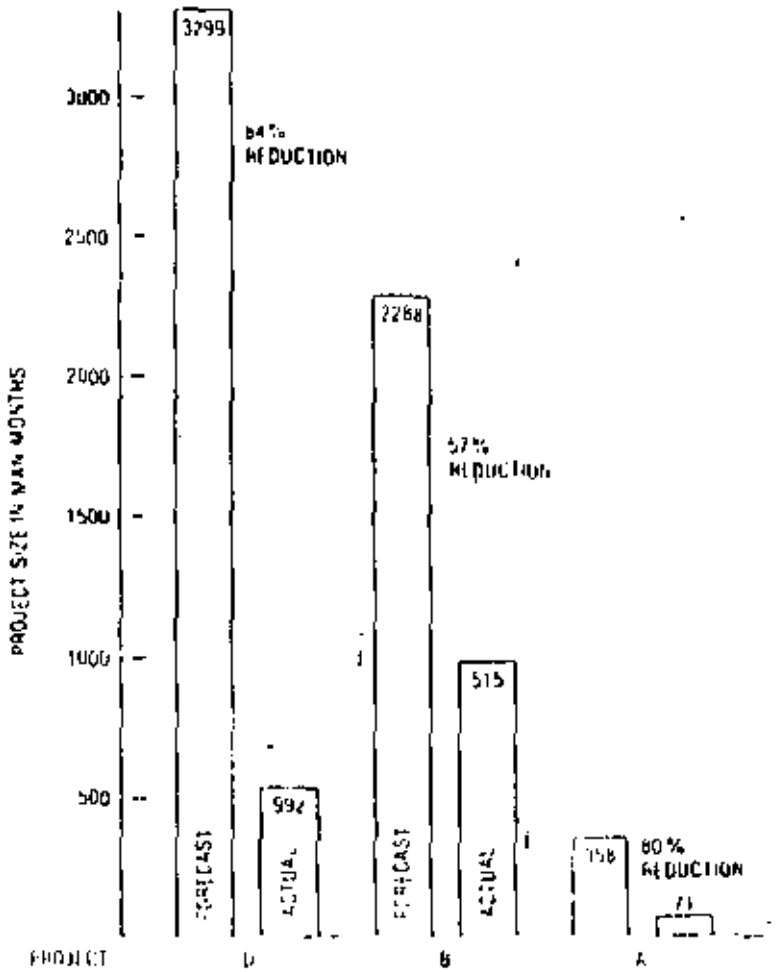


Figure 1 Improvement of actual man-months over forecast man-months on three Boeing software projects as a result of adopting modern programming practices.

Table 3. Modern programming practices used on three Boeing projects (listed in order of impact on cost).

PRACTICES ASSOCIATED WITH PROGRAM MANAGERS' MANAGEMENT METHODS (USED BY ALL THREE PROJECTS)
WRITTEN TASK ASSIGNMENTS
FORMAL CUSTOMER REVIEWS
EARLY DOCUMENTATION
UNIT DEVELOPMENT FOLDERS (PROGRAMMERS' WORKBOOK)
DESIGN REVIEW PRIOR TO CODING (STRUCTURED WALKTHROUGH)
FORMAL TESTING (USED BY TWO LARGER PROJECTS ONLY)
CONSTRUCTION PLANNING (INCLUDING PLANNING FOR INTEGRATION AND FUNCTIONAL TESTING)
CODE VERIFICATION (PEER CODE REVIEWS)
ACCEPTANCE TESTING (FORMAL DEMONSTRATION)
FUNCTIONAL TESTING
CONTROL OF TEST MATERIALS
CONFIGURATION MANAGEMENT (USED BY TWO LARGER PROJECTS ONLY)
BASELINING
PROBLEM REPORTING
CHANGE CONTROL BOARD

revealed universal positive feelings about the effectiveness of top-down design techniques:

> Probably the improved customer/project and intra-project communication cited by our interviewees as a consequence of top-down design has its primary benefit in establishing a sound basis for formal testing. In particular, two of the program managers interviewed felt that their testing activities proceeded more smoothly, and that fewer errors were discovered during testing as a direct result of the top-down design techniques they employed. For lack of supporting data in this study, we can only conclude that the cost benefits of top-down design may not be evident until a software system is in operation and maintenance.

On the other hand, the absence of a positive correlation between modern programming practices and cost on the two smaller projects was more likely the result of the idiosyncracies of small projects than it was of the employment of top-down design practices. The positive correlations found on the three larger projects, which also used top-down design and the other techniques, is more persuasive.

Black also attempted to measure the impact of modern programming practices on the percentage distribution of costs over the four main project phases used by Boeing:

- definition (essentially requirements and specifications)
- design
- construction (includes coding, integration, and functional testing)
- demonstration (acceptance testing and installation)

The results, summarized for the three projects in Figure 2, showed that costs were shifted into earlier stages by the use of modern programming practices. Unfortunately for the study design, Boeing included both coding and testing in the third stage, construction. Consequently, the shift to the left was less marked than it might have been if test costs were broken out separately.

Programming standards. In the first of two related studies, Brown[9] interviewed a cross section of management and performer personnel on TRW's very large ballistic-missile-defense programming project, as well as key staff personnel outside the project. This survey covered the impact of 16 detailed programming standards (e.g., structured coding) on 30 characteristics of software or the development process (e.g., cost, schedule, or programmer productivity). That made a matrix of 540 relationships, each of which was categorized over some nine levels of combined influence and assertion strength from very strong positive to very strong negative. However, 43 percent of the 540 intersections were labeled indifferent or inconclusive, implying either that there really was very little relationship between these variables or that the interviewees were ignorant of them.

As a brief summary of some of Brown's findings, four of the more significant rows (representing various coding standards) and seven of the more meaningful columns (representing various software characteristics) have been brought together in Table 4. Brown's ratings (strong positive, for example, means strong assertion, positive influence) were converted to weighted integers (strong positive = +3), both for simplicity and to permit the values to be summed across the rows. Thus, routine size or modularity is related to the seven software characteristics to the degree of 57 percent of the maximum positive rating. It is related to all 30 software characteristics (not shown in Table 4) only to the degree of 28 percent. This drop is not surprising, since the seven columns, with one exception, were selected to show the stronger relationships. Structured coding, which was not strongly correlated, was included as a matter of interest, since it is one of the core practices.

Structured coding evoked strong feelings on the part of the interviewees. Overall it had 15 positive ratings (average +1.7) and seven negative ratings (average −2.86). As Brown pointed out, this was seven out of a total of only 17 negative ratings in the whole survey. He felt that it "indicates a relatively dim view of structured coding on the part of [project] personnel."

"However," he went on, "there are some good reasons for this to be the case. First, the standard was not explicitly defined and enforced until almost two years after [the project] began, and the requirement to restructure existing code was felt to be counterproductive. Moreover, writing structured code in standard Fortran is awkward and introduced some inefficiencies in core and execution time making it difficult to satisfy demanding requirements levied on the real-time software. Finally, [the project] has not yet reached the phase during

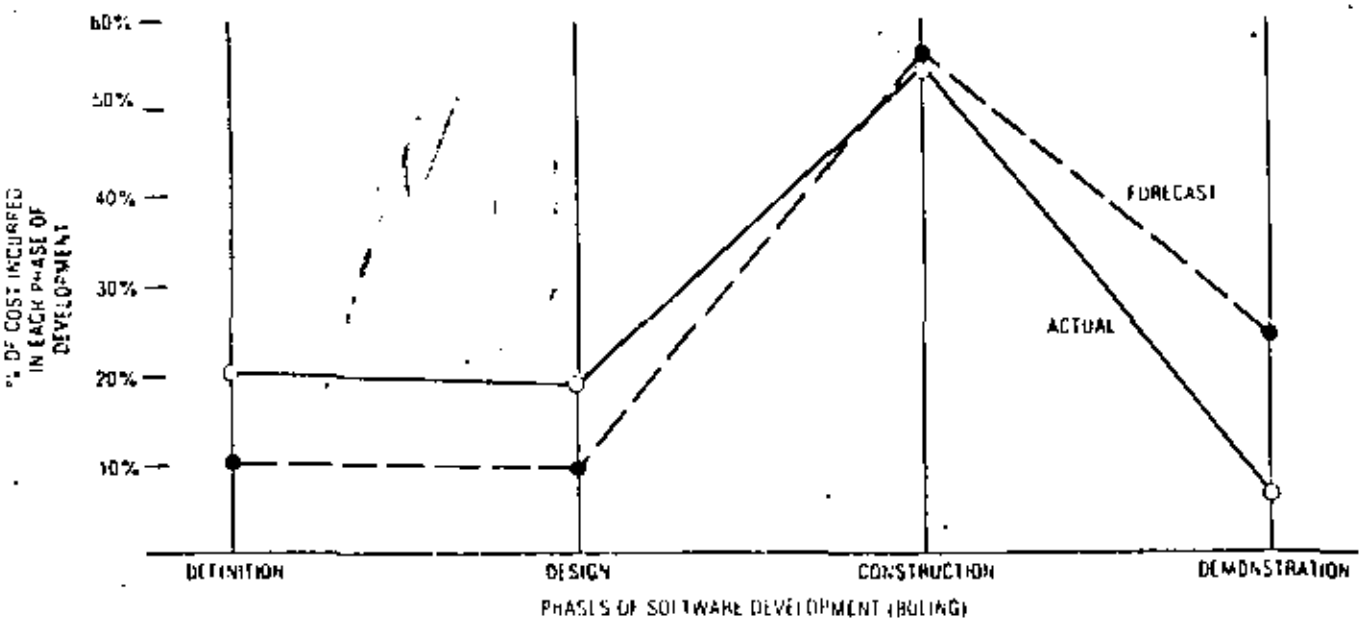


Figure 2. Forecast cost distribution shifts to earlier phases when modern programming practices are employed (average of three projects).

Table 4. Relationship of selected programming standards and software characteristics (first TRW study)

PROGRAMMING STANDARDS	SOFTWARE CHARACTERISTICS								
	CODE AUDITABILITY	CODE UNDERSTANDABILITY/ READABILITY	CODE MAINTAINABILITY/ USABILITY	TESTABILITY	OPERATIONAL RELIABILITY	CODING ERROR FREQUENCY	PROGRAMMER PRODUCTIVITY	TOTALS FOR 7	TOTALS FOR 30
ROUTINE SIZE (MODULARITY)	+3	+3	+3	+3	+2	+2	0	+16 +57%	+34 +26%
IN-LINE COMMENTARY	+3	+4	+4	+1	+1	+2	0	+15 +54%	+37 +31%
STRUCTURED CODING	+3	+3	0	+2	+2	0	−1	+7 +25%	+11 +9%
NAMING CONVENTION	+2	+3	+2	+1	+1	+2	+2	+13 +46%	+38 +32%

which the major benefits of structured programming (i.e., improved readability and maintainability) were expected to be reaped."

On the relationships between coding standards and the characteristics of cost, schedule, and programmer productivity, the study was two-thirds inconclusive and one-third negative. That is, about two-thirds of the intersections for these categories were labeled inconclusive or indifferent. The remaining one-third of the intersections were mostly negative, showing the following net percentages:

Cost	Schedule	Programmer Productivity
−33 percent	−25 percent	+7 percent

Perhaps the reasons for the poor showing on these characteristics are similar to those quoted above on structured coding.

In summation, the overall weighted ratings for 18 programming standards against 30 software characteristics were 59 percent positive, 36 percent indifferent or inconclusive, and 5 percent negative.

In addition, two hypotheses were tested. The first is that programming (i.e., documentation and coding) standards, if rigorously defined, systematically enforced, and supported by tools, help to make possible the production of software of higher than usual quality. To this proposition those interviewed agreed 85 percent, disagreed 4 percent, and

didn't know 11 percent. To the second hypothesis—i.e., that these standards help to make possible the production of software of lower than usual cost—those interviewed agreed only 28 percent, disagreed 50 percent, and didn't know 22 percent.

MPP impact. The second TRW evaluation survey generated a matrix of 11 modern programming practices against 12 software problems, making 132 intersections in all. In this study personnel were asked to respond on both an *actual* and a *theoretical* basis. The theoretical responses showed a higher degree of relationship than the actual by a weighted margin of better than 3 to 2. In addition, the "indifferent" or "inconclusive" intersections dropped from 40 to eight. Of the 40 indifferent or inconclusive intersections in the actual comparison, 26 were involved in cost or schedule overruns or inefficient use of resources, again suggesting the interviewees were currently dubious in these areas. On a theoretical basis, however, the indifferent and inconclusive intersections to these three software problems dropped to five, implying that with more experience, modern programming practices would have a more favorable impact on these problems. In fact, the ratings for these three problems increased from "inconclusive" to "medium" or "strong positive."

The average impact of each modern programming practice on the twelve software problems is graphed in Figure 3. The theoretical ratings are considerably better, this time by about 2 to 1 in weighted terms.

In Brown's own words, "There is strong agreement among [project] personnel as to the four MPP of greatest importance and impact and strong agreement on their relative ranking:

- Requirements analysis and validation
- Baselining of requirements specification
- Complete preliminary design
- Process design

There is strong agreement on the importance and positive impact of the next three most highly ranked MPP, but the relative ranking among them is less clear:

- Incremental development
- Unit development folders
- Software development tools

There is strong agreement on the importance and positive impact of the four lower ranked MPP, but the relative ranking among them is not at all clear.

- Independent testing
- Enforced programming standards
- Software configuration management
- Formal inspection of documentation and code."

As to the general MPP hypothesis, the query and the results were as follows: Rules governing software development, evaluation, and documenta-

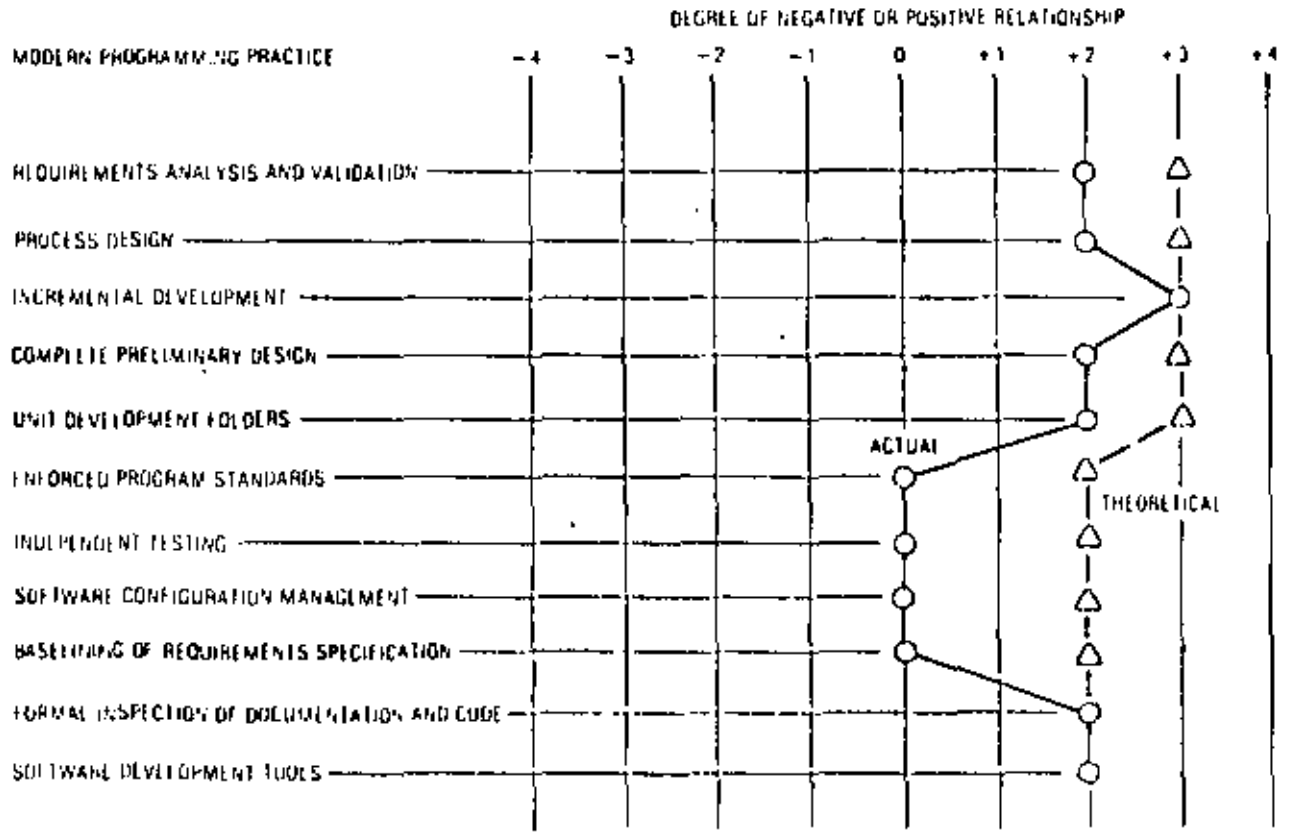


Figure 3. The correlationships between the modern programming practices and the summation of the twelve software problems is stronger in the theoretical case than in the actual case by about two to one

tion, if rigorously defined and applied and supported by modern programming practices (techniques and tools), make possible the production—

	True	False	?
—of higher than usual quality software	65 percent	4 percent	31 percent
—of lower than usual life cycle cost	49 percent	15 percent	36 percent

Error relation. Operational reliability, documentation error frequency, and coding error frequency were three of the 30 software characteristics employed by Brown in his first study. The weighted positive rankings were as follows:

Operational reliability	31 percent
Documentation error frequency	14 percent
Coding error frequency	31 percent

In his second study, he found a more positive relationship between eleven modern programming practices and reliability, as follows:

Actual:	57 percent
Theoretical:	73 percent

Belford, Donahoo, and Heard[9] used software error as the only criterion in their study of 21 technical and five management software engineering techniques. In the absence of firm historical records, data was obtained by interviews with experienced personnel. The basic data was in two categories:

- the percentage of total errors estimated to occur in each of seven phases of the software life cycle, however, only the distribution of errors in the code and checkout phase was employed in this study;
- the probability of detection of each of the twelve error types by each of the 26 software engineering techniques.

This interview data was processed to obtain index numbers ranging from 0 to 75. These numbers indicate the effectiveness of each software engineering technique in reducing errors, as summarized in Table 5.

The authors regard their present method as the prototype of a still unproved process that "can reduce the identification and selection of optimum software engineering techniques to a straightforward, well defined procedure."

To carry the procedure further, better historical data is needed, as well as information on other phases of the life cycle.

The broader community

So, members of the defense industry are making headway in the effort to embrace software development in engineering discipline. But what of the broader data processing community—the banks, insurance companies, and commercial establishments? Not long ago, Harlan Mills felt still able to say: "The idea of a rigorous rather than a heuristic program design method is new, and is still largely unknown in programming as practiced today."[4]

The IBM approach. Many of the improved programming technologies were developed by IBM scientists—Mills for one. The big computer maker has been a leader in their application, both within the company and to its customers. One task the industry faces, according to one IBM executive, is the need for a concerted effort by all elements—the manufacturers, the service bureaus, software houses, schools, and universities—to bring within their own training programs or curricula instruction in the improved programming technologies.

The increase—the delta—of programmers being added to the industry can then be addressed through the classical education channels. A bigger problem, with knowledge in the field moving so rapidly, is recycling established professionals—bringing their skills up to the latest levels of knowledge. In this area IBM has included the new programming practices in its educational offerings to customers. It is also working to instill a knowledge of these practices into its own very extensive programming population.

Table 5. Effectiveness of software engineering techniques in detecting errors (based on code and checkout phase only). Techniques are listed in order of effectiveness, with management techniques in parenthesis.

VERY EFFECTIVE (INDEX NUMBER 45 TO 75)
- PROGRAM REVIEWS
- CHIEF PROGRAMMER
- CHIEF PROGRAMMER TEAM
- BUILD LEADER (A COMPUTER SCIENCE CORP. TECHNIQUE)[7]
- STRUCTURED WALK-THROUGHS

EFFECTIVE (28-33)
- INDEPENDENT TEST AND EVALUATION
- EXECUTION ANALYSIS
- PROGRESSIVE TESTING

LESS EFFECTIVE (10-16)
- PROGRAMMING TECHNIQUES
- STRUCTURED PROGRAMMING
- TOP DOWN PROGRAMMING
- VERIFICATION PROCEDURES
- SUPPORT PROGRAMS
- (TOP DOWN DEVELOPMENT)

LEAST EFFECTIVE (0-4)
- (INSPECTION TEAMS)
- TOP DOWN DESIGN
- STRUCTURED DESIGN
- BUILDS (A CSC TECHNIQUE)
- (TRIBADS MANAGEMENT SYSTEM) (A CSC TECHNIQUE)
- PROGRAMMING DESIGN LANGUAGE
- AUTOMATED NETWORK ANALYSIS
- (CHIEF'S REVIEWS)
- (MANAGEMENT DATA COLLECTION)
- PROGRAMMING SUPPORT LIBRARY
- SOFTWARE CONFIGURATION MANAGEMENT
- PROGRAMMING LIBRARIANS

JPL Improves Software Development Process

At Caltech's Jet Propulsion Laboratory, about one fifth of the budget and one sixth of the manpower are devoted to some aspect of computing. All the software development organizations, including the Mission Control Center, Deep Space Network, spacecraft on-board computing, spacecraft testing, and the centralized computing facilities, have shown a keen interest in systematic design methods and, beginning early in the 1970's, in structured programming, top-down development, and other modern programming practices.

Management of JPL deliberately took a low-key approach to the introduction of these practices. At first there was a minimum of formal directives and a maximum of suggestion and example. The programming librarian idea was transformed by the Deep Space Net into a means of assisting programmers called the "programming secretariat." In the last several years the Deep Space Net has issued a set of rigorous software development standards, as listed in the accompanying box. In addition, a JPL staff member, Robert C. Tausworthe, prepared a 379-page monograph which presents these practices in a logical, tutorial form.[1]

JPL took a low-key approach to introducing the new techniques.

Flight projects and the Mission Control Center, which is shared by the projects, have not specified software development methods to the same level of detail as the Deep Space Net. The particular way in which these methods are implemented is adapted by each project organization to the tools available on the computer used for each task. However, the use of a program design language, structured programming, top-down development, and related methods are de facto standards in the project areas. Since much of the programming is scientific or engineering in content, an early task was the development of SFTRAN, Structured Programming-To-Fortran Translator. Recently a preprocessor to facilitate structured programming in assembly language has been developed.

So, although the laboratory does not insist upon completely standardized organization-wide programming practices, the underlying principles are common to all sections. The differing detailed procedures endorsed by various program offices are the result of differences in the kind of applications. For example, the Deep Space Network develops software to be operated at overseas sites by personnel who are not JPL employes but are permanent residents of the countries in which the stations are located. Conversely, flight projects utilize the same personnel who develop the software to operate, maintain, and modify it during the mission. Moreover, flight software is much more subject to frequent modification in response to changes in the mission than is the software in the Deep Space Net.

Personnel. By 1975 an informal census of a cross section of programmers found about 50 percent definitely favorable to the new practices, about 40 percent rather neutral, and about 10 percent hostile to them. However, it was not only the modern programming practices that were important in forming these attitudes, but also the environment in which the work was done—the accessibility of text editing, interactive computing, and tools and facilities that enabled the programmer to get his job done with a minimum of running around and standing in line. They like the programming secretariat and support concepts. This environment made their lives easier and more productive.

Still there is great variability in the individual productivity of programmers, perhaps by a factor of 10 to 1, or even more sometimes. The techniques of structured design do not, of course, change dunces into brilliant programmers. They do make it possible to break up a large project in such a way that the more competent personnel can be assigned to the difficult early stages and

Software Standard Practices

DEEP SPACE NETWORK STANDARD PRACTICES
- 810-13 (AUGUST 15, 1975) SOFTWARE IMPLEMENTATION GUIDELINES AND PRACTICES
- 810-16 (DECEMBER 15, 1975) PREPARATION OF SOFTWARE REQUIREMENTS DOCUMENTS
- 810-17 (JULY 15, 1976) PREPARATION OF SOFTWARE DEFINITION DOCUMENTS
- 810-19 (MARCH 1, 1977) PREPARATION OF SOFTWARE SPECIFICATION DOCUMENTS
- 810-20 (FEBRUARY 1, 1977) PREPARATION OF SOFTWARE OPERATOR'S MANUALS
- 810-21 (NOVEMBER 15, 1976) PREPARATION OF SOFTWARE TEST AND TRANSFER DOCUMENTS

PROJECT DOCUMENTS
- PD/ITR 58 (November 13, 1974) MARINER JUPITER/SATURN 1977 SOFTWARE MANAGEMENT PLAN (REVISED SEPTEMBER 7, 1976)
- PD/22-35 (SEPTEMBER 7, 1977) SEASAT-A ADF PROGRAMMING STANDARDS

GENERAL SOFTWARE DOCUMENTS
- NASA SOFTWARE MANAGEMENT GUIDELINES
- JPL PUBLICATION 77-24 (JULY 1, 1977) SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE, BY HENRY FLEINE
- INTEROFFICE COMPUTING MEMORANDUM NO. 337 (JULY 31, 1973) SFTRAN USER GUIDE (STRUCTURED PROGRAMMING TO FORTRAN TRANSLATOR) BY JOHN A. FEVIGA
- COMPUTING MEMORANDUM 432 (NOVEMBER 11, 1977) PROGRAMMING FOR RELIABILITY, BY FRED T. KROGH, CHARLES L. LAWSON, AND MICHAEL R. WAGNER

the less creative people can be assigned to implement tasks that have been fairly well structured. The structured methods also enable managers to identify the real incompetents sooner than older methods did.

Advantages. The payoff from the adoption of modern programming practices has come in terms of significantly improved schedule performance and manpower productivity. There is no doubt that these techniques have resulted in dramatic improvements in cost, quality, and schedule. For example, one of the programs for the Voyager project—several hundred thousand lines of high level language—was recently completed with labor expenditures within budget and computer time at something like half of budget. The reduction in computer time implies that the programs were of better quality, because the programmers had to do much less debugging and testing. In the entire Voyager software development program, just finished, only one "tiger team" had to be set up. That was in the data records area which had to be done in assembly language because a higher-level language was not available.

A recent Deep Space Net project, one that had been budgeted for two years, came in two weeks over budget, but this overage had been detected a year ahead and appropriate readjustments had been made. In this case, too, debugging and testing took half the time originally budgeted. The reason: very few errors. In fact, there were 350 errors, including those in documentation, in 300,000 lines of code—a rate of only 20 percent of previous experience. Perhaps of equal importance to those only too accustomed to nerve-wracking "tiger team" efforts to meet flight dates, this time there was no tiger team. There were no hassles and no one aggravated his ulcers. It seemed that software management was coming of age.

Moreover, the full extent of the savings from the use of the new practices will only become apparent several years into the maintenance phase. This phase is clearly going to be much more efficient. The programs will be more reliable, they will be easier to change when necessary. These improvements follow from the experience in the implementation phase. When this phase is completed more quickly and with fewer errors, there is more confidence that the program itself is closer to full correctness.

JPL is finding a greater percentage of the overall software development effort being concentrated at the front end under the new practices. Coding is expected to remain a small fraction, even as the proportion of resources devoted to testing and maintenance declines. The 40/20/40 percent rule for the division of resources between definition and design, coding, and testing generally represents the laboratory's experience.

Still, there is an element of management in the problem. There is a need for disciplined thinking. Rather than focus on the fact that more money is being spent on software development, the big users have tended to emphasize fine tuning. Lately there does seem to be an increasing awareness by management of the dichotomy between the price/performance improvement of the hardware and the lack of that kind of improvement in software development. In the final analysis the users themselves have got to be willing to devote effort and money to improving their software development process.

Productivity relation. Although Brown found little relationship (+7 percent) between his 18 coding and documentation standards and programmer productivity, another approach showed a significantly high correlation. Walston and Felix[9] set up a software measurement project in the IBM Federal Systems Division in 1972. One of its purposes was to assess the effects of structured programming, then just beginning to be used, on the software development process. Data from 60 completed projects is now in their data base, representing projects ranging from 4000 to 467,000 source lines and 12 to 11,578 manmonths. Source lines were defined as the input to a language processor.

Using this data base, 68 variables were analyzed and 29 showed a high correlation with productivity.

The overall judgment is: modern programming practices are ready for use.

The data available enabled a comparison to be made between delivered source lines per manmonth and the percentage of code developed using each technique. These relationships, visualized in Figure 4, showed a rate of productivity improvement averaging 70 percent between "little use" of the new techniques and "much use."

Real-time design. The Advanced Technology Center of the Army's Ballistic Missile Defense has been engaged since 1972 in identifying and refining techniques to improve the software designer's ability to program large real-time processes in which the timing and order of events are critical. The result of its efforts, called process design methodology, embraces structured programming, top-down development, and other methods.

Gaulding and Lawson[10] analyzed a data base collected during experimental development work containing over 10,000 labor and computer-run entries. Two distinct advantages of the new design methodology over conventional approaches emerged.

- a majority of software errors was detected prior to the 50 percent project completion point, the opposite of conventional experience, and
- productivity was 33 equivalent machine instructions per manhour, said to be "considerably

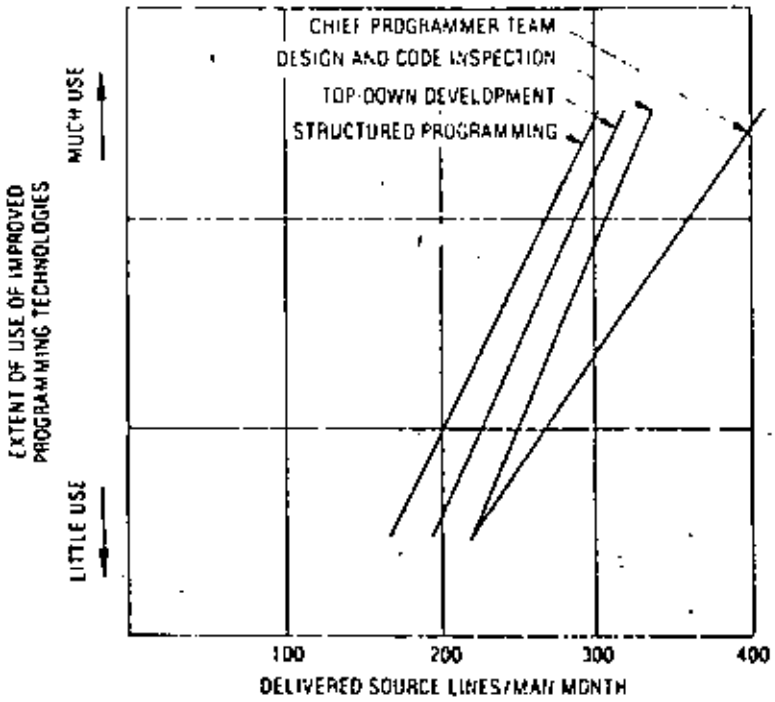


Figure 4. Production appears to increase dramatically with the use of improved programming technologies.

better" than the results on comparable real-time projects programmed by conventional methods.

Data management. Hsiao[11] reported the development of a highly secure data management system composed of 80 modules with 40K words of code. The total design and implementation were completed in 3 manyears with an elapsed time of less than 11 months. Using a chief programmer team, structured programming, and the other concepts of modern programming practices, he felt that the effort could not have been completed in such a short period of time without the use of these techniques. In addition, said Hsiao, extension of the system "has been greatly facilitated by the complete development-support library, clear system interface, high program modularity, and well-structured code."

Japanese findings. The Nippon Electric Company has modified its data base management system, programmed in Cobol, to take advantage of the concepts of structured programming.[12] This experience, plus some experiments in using the new techniques, has convinced them that the methods can be successfully applied to the forthcoming development of a very large on-line banking system. Their experiments indicate the following:

- number of steps programmed per working hour doubled;
- number of steps programmed per machine hour used tripled;
- number of database handling errors decreased 30 percent;
- other types of programming errors decreased 65 percent;
- average length of a bug find-and-fix cycle has been reduced to about one-third the conventional time.

These improvements were accompanied by small degradations in program performance.

Early assessment. Back in 1974, after only a few years of experience with the group of techniques that were then called structured programming, the IEEE Computer Society's Lake Arrowhead Workshop found that savings of "over 50 percent had been achieved, relative to previous performance on similar projects."[13] Companies providing data included IBM Federal Systems Division (40 percent average improvement over 20 projects), McDonnell-Douglas Automation Company (36 percent improvement on three projects; 5 percent on a fourth), and Hughes Aircraft Company (50 percent improvement on two real-time projects).

New practices judged effective

Software is in a predicament: it is too costly, too error-prone, too complex, too hard to maintain. This predicament may delay new applications of computers unless the effectiveness of software development can be substantially improved.

Modern programming practices are not the only way, of course. Higher-level programming languages, improved life cycle planning and management,* automated development tools, better personnel selection and training, and, of course, more sophisticated management will undoubtedly help.

The studies brought together here indicate that software effectiveness can be achieved. Where the results have been reduced to numbers, they ranged from about 25 percent to about 75 percent on such factors as cost, error reduction, and productivity.

Too much weight should not be placed on any one number. Studies of this kind are difficult to make and are usually not comparable from one organization to another, because definitions of terms vary. Moreover, averaging various amounts of knowledge—or degrees of ignorance—from interviews does not necessarily provide precise numbers. And summarizing detailed results, as has been done in this article, almost inevitably obscures the nuances of the original papers.

Rather, it is the overall judgment that is impressive: Whether from formal studies or the impressions of experienced executives, it seems clear that modern programming practices are effective in improving the processes of software development. They are ready for use. ■

*An article on life cycle planning is scheduled for the second quarter of 1978.

Acknowledgments

Interviews with the following experienced observers of software development helped guide me through the literature and practices of this field:

Walter R. Beam, Office of the Assistant Secretary of the Air Force for Research, Development, and Logistics

Joe M. Henson, IBM Data Processing Division

Robert Jirka, Michael R. Plesset, Edward C. Posner, and Michael R. Warner, all of Jet Propulsion Laboratory, California Institute of Technology.

References

1. Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays On Software Engineering*, Addison-Wesley Publishing Co., Reading, Massachusetts, 195 pp., 1974.

2. F. T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Sys. J.*, 11,1 (1972), pp. 56-73.

3. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972, 220 pp.

4. Harlan D. Mills, "Chief Programmer Teams, Principles, and Procedures," IBM Federal Systems Division Report FSC 71-5108, Gaithersburg, Maryland, 1971.

5. Harlan D. Mills, "Top Down Programming In Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin (ed.) Prentice-Hall, Englewood Cliffs, New Jersey, 1971, pp. 41-55.

6. Robert M. McClure, "Software—The Next Five Years," *Digest of Papers, COMPCON Spring 76*, pp. 6-7.

7. John B. Holton, "Are The New Programming Techniques Being Used?" *Datamation*, July 1977, pp. 97-103.

8. Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.

9. Walter R. Beam, "Can Software Be More Like Hardware? Should It Be?" Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*).

10. Edward W. Pullen and Robert G. Simko, "Our Changing Industry," *Datamation*, January 1977, pp. 49-55.

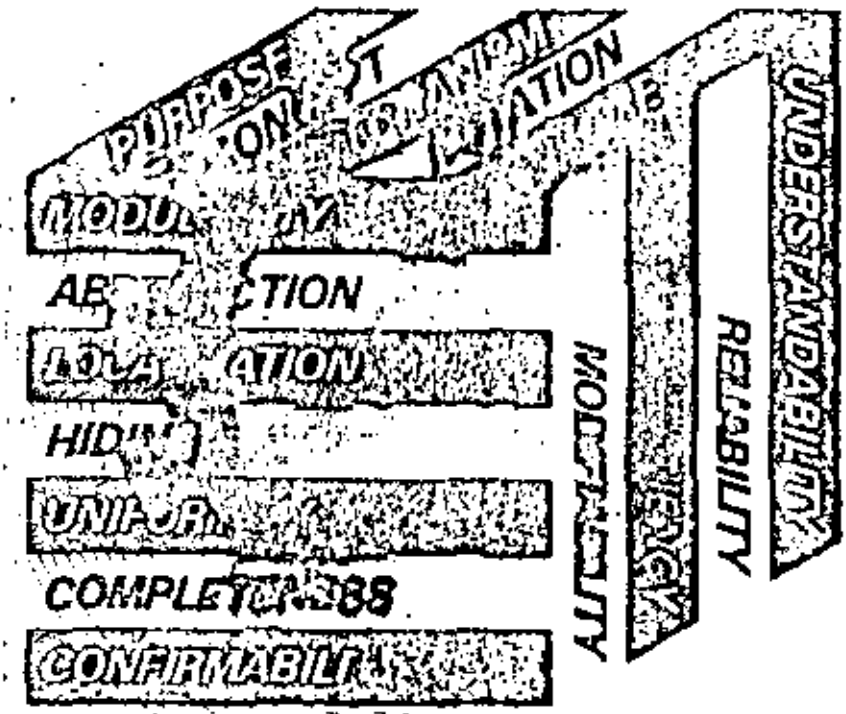
11. Joe M. Henson, "Computer Applications, Trends and Directions," Keynote speech at COMPCON Fall 77, September 7, 1977 (not in *Digest of Papers*).

12. "Business Software Makes For Problems, Hosier Charges," *Electronics*, November 24, 1977, p. 14.

13. J. L. Elshoff, "An Analysis of Some Commercial PL/I Programs," *IEEE Trans. Software Engineering*, June 1976, pp. 113-120.

14. N. French, "Programmer Productivity Rising Too Slowly: Tanaka," *Computerworld*, 1977, 11 (32) p. 1.

This paper attempts to define the principles and goals that affect the practice of software engineering. Its intent is to organize these aspects of software engineering into a framework that rationalizes and encourages their proper use, while placing in perspective the diversity of techniques, methods, and tools that presently comprise the subject of software engineering.



SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS

Douglas T. Ross, John B. Goodenough, C.A. Irvine
SofTech, Inc.

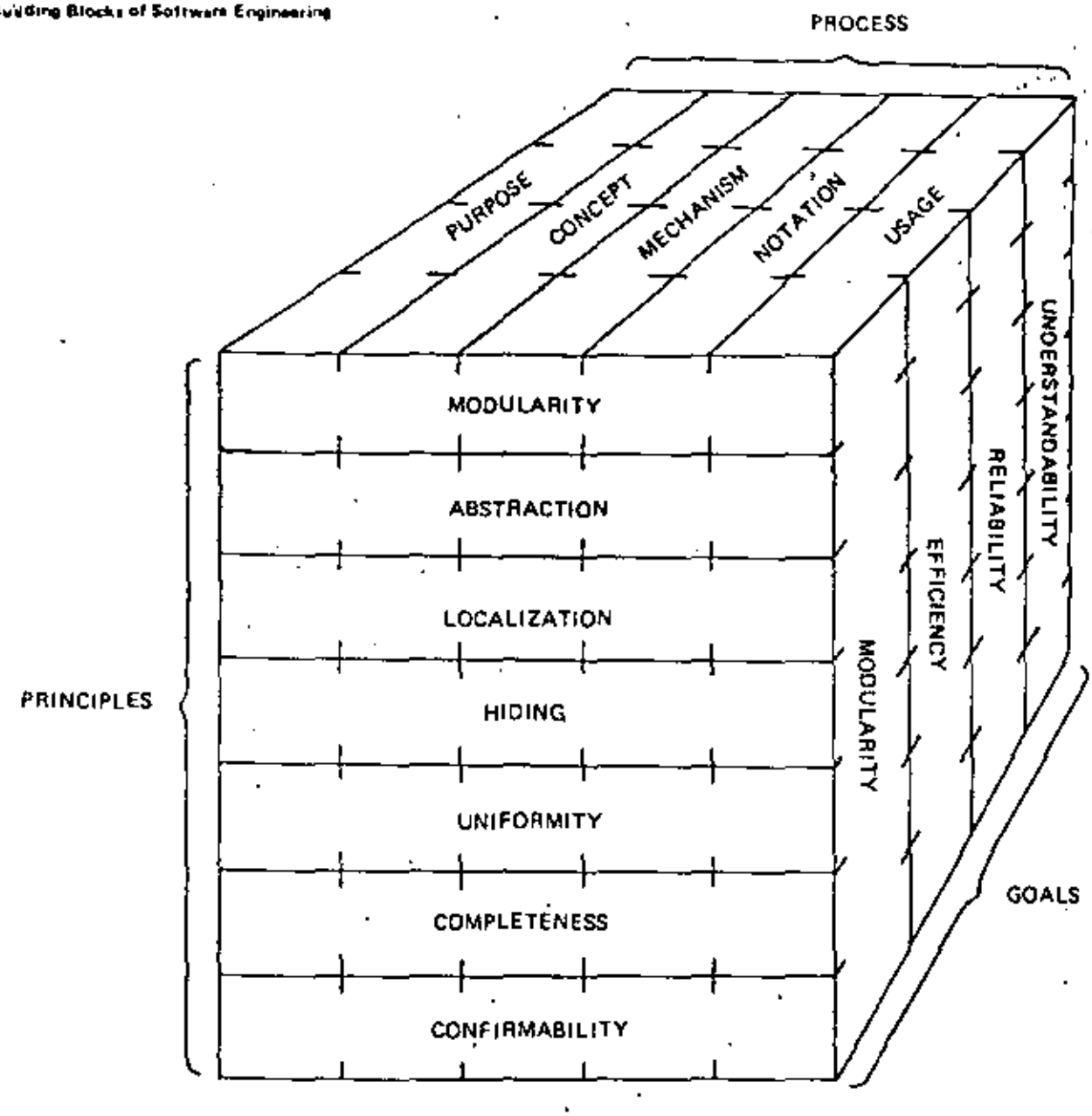
Introduction

The conferences sponsored by NATO in 1968 and 1969 gave popular impetus to the term "software engineering." Since that time the need for a more disciplined and integrated approach to software development has been increasingly recognized. Although useful definitions of the term remain elusive, software engineering clearly implies at least the disciplined and skillful use of suitable software development tools and methods, as well as a sound understanding of certain basic principles. In this paper, we attempt to expound what these principles are, and how they are applied in the practice of software engineering.

It is perhaps best to view this paper as an attempt to identify the important underlying issues of software engineering in a form that permits the interaction of these issues to be better understood. We will discuss these issues in terms of four fundamental goals: *modifiability*, *efficiency*, *reliability*, and *understandability* as well as seven principles that affect the process of attaining these goals:

- the *modularity* principle, which defines how to structure a software system appropriately;
- the *abstraction* principle, which helps to identify essential properties common to superficially different entities;
- the *hiding* principle, which highlights the importance of not merely abstracting common properties but of making inessential information *inaccessible* (hiding deals with defining and enforcing constraints on access to information);

Figure 1. The Building Blocks of Software Engineering



- the *localization* principle, which highlights methods for bringing related things together into physical proximity;
- the *uniformity* principle, which ensures consistency;
- the *completeness* principle, which ensures that nothing is left out;
- the *confirmability* principle, which ensures that information needed to verify correctness has been explicitly stated.

These principles and goals are applied in the practice of software engineering, which deals with various software development activities:

Determine requirements–the process of identifying the requirements to be satisfied by a software system; the objective is to define the problem to be solved in terms of the constraints a solution must satisfy, including cost and performance.

Design software–the process of considering each user requirement and creating the conceptual basis on which the problem is to be solved; design is the process of deciding *how* to satisfy user requirements within the allowed constraints.

Specify implementation–the process of describing the interactions between the designed modules of a solution; the result of this activity is a detailed specification of constraints the software implementation must satisfy, but not the software itself.

Code/debug–the process of actually producing the software satisfying the specification, and verifying that the produced software does satisfy the user requirements.

Tuning–the process of modifying a logically correct system until it meets performance goals.

Despite the obvious differences among these activities, we believe each reflects a common pattern which we call the *fundamental process*. This process consists of five basic steps: (1) crystallize a *purpose* or objective; (2) formulate a *concept* for how the purpose can be achieved; (3) devise a *mechanism* that implements the conceptual structure; (4) introduce a *notation* for expressing the capabilities of the mechanism and invoking its use; (5) describe the *usage* of the notation in a specific problem context to invoke the mechanism so the purpose is achieved.

This sequence of steps has a natural parallel to the process of software development:

purpose – define the requirements for a system;

concept – derive the architecture of a software system to satisfy these requirements and specify the modules that constitute the system;

mechanism – implement the software system (devising the mechanism is obviously the code/debug/tune activities in the development process);

notation – define the command language or other means a user will employ to invoke the capabilities of the software system;

usage – describe how the software system is controlled (this description may take the form of a user's manual for the system).

Equally well, the pattern defined by the fundamental process could be applied differently, to highlight a different aspect of software development. For example, we could leave the description of purpose and concept as above, but replace mechanism, notation, and usage with the following descriptions:

mechanism – the computer on which the software runs;

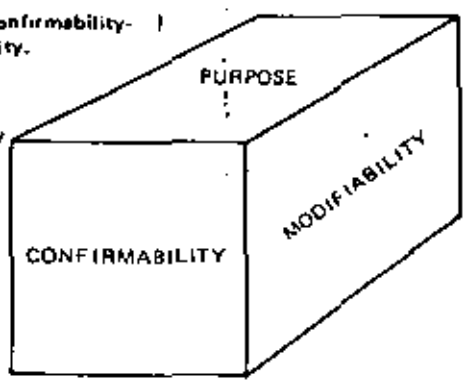
notation – the programming language in which the software will be written;

usage – the manual describing how the language is used to control the computer.

The interpretation of the fundamental process is clearly highly context-dependent. It is also intimately tied to the notion of *hierarchical decomposition*—the widely recognized phenomenon of part/whole relationships. A purpose is composed of sub-purposes; a mechanism has many parts which are themselves sub-mechanisms, and in general, the mechanism itself is only a part of some super-mechanism, etc. The interesting phenomenon is that *both* the pattern of the fundamental process *and* part/whole hierarchical relationships must be employed in all aspects of software engineering practice.

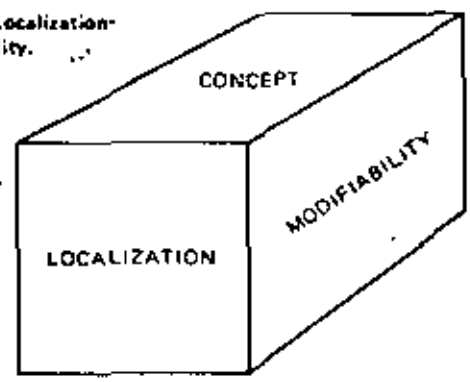
Given that part/whole hierarchical decomposition plays a pervasive role, the fundamental process interacts further with the various principles and goals of software engineering as depicted in Figure 1. Figure 1 is our framework for discussing the nature and issues of software engineering. Clearly an exhaustive treatment is not possible, but the remainder of this paper is intended to show how the structure of Figure 1 does yield insight into the theory and practice of software engineering. By way of illustration, consider the following examples (Figures 2-7) of how principles, goals, and process elements interact:

Figure 2. Purpose-Confirmability-Modifiability.



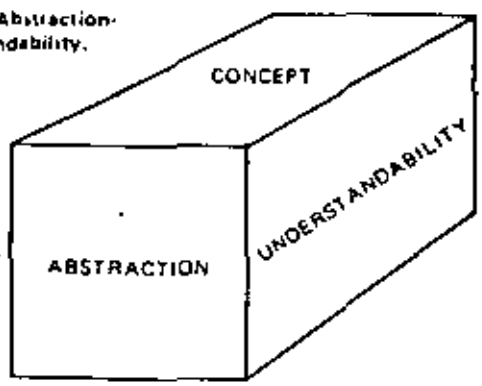
- The purpose of a design choice may be to structure a system so the effects of a change can be predicted with assurance.
- Confirmability applied to the process of defining purposes for modifiability demands that purposes be stated in a form that makes it possible to easily check if they have been achieved, e.g., an objective whose achievement would enhance modifiability would be to insure that only declarations in a program need be changed when transferring a program to a new computer. This is a confirmable statement of objectives while "reducing the number of changes that must be made" is not so clearly confirmable, and hence, is not so useful.

Figure 3. Concept-Localization-Modifiability.



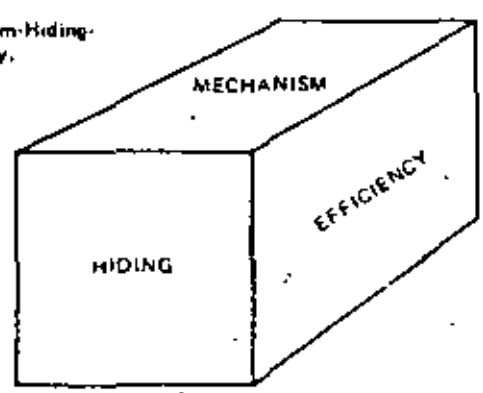
- Identifying one module for implementing a scheduling policy in an operating system is an example of the effect of the localization principle on design concepts to enhance modifiability, since localizing the policy rather than scattering policy decisions throughout a system makes it easier to change the policy.
- Having decided on the previous design for scheduling, the concept of a table-driven scheduler, which localizes scheduling policy in a single table *within* the module, then makes the *module* more modifiable. These two examples illustrate the role of hierarchy in applying the principles, goals, and process steps.

Figure 4. Concept-Abstraction-Understandability.



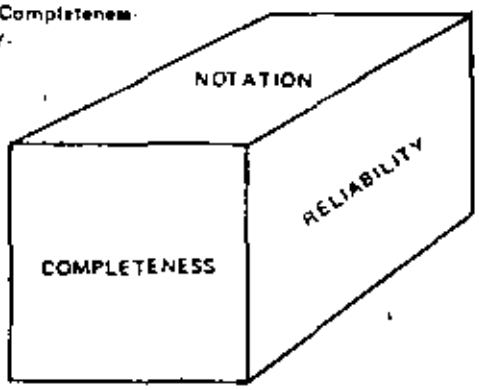
- The use of levels of abstraction[5,11,10] to define an understandable program or system of programs shows how abstraction can make designs more understandable.
- High order languages represent a concept for improving the understandability of programs by abstracting from the details of computer instruction sets

Figure 5. Mechanism-Hiding-Efficiency.



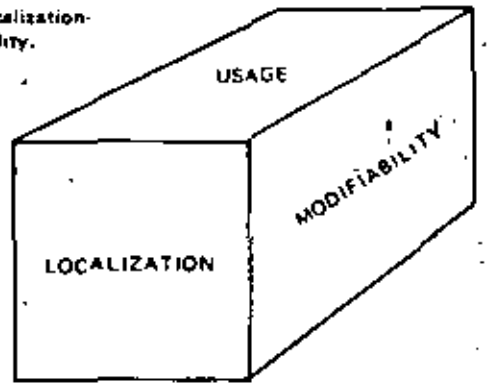
- Knowing that only certain subroutines have access to shared data (e.g., as in a function cluster[12]) may permit fewer checks to be made on the validity of stored values, and thereby increase efficiency.
- Forbidding users from directly accessing I/O devices permits an operating system to optimize overall usage of the devices.

Figure 6. Notation-Completeness-Reliability.



- Having a complete set of convenient goto-free control structures is necessary to avoid error-prone circumlocutions.
- In designing a case statement, if the form does not permit a programmer to specify what is to happen when the case statement variable is out of range, then the programmer is unable to easily treat this possibility, and hence is not encouraged to develop error recovery algorithms. A complete notation can foster reliability by permitting a programmer to specify error recovery details.

Figure 7. Usage-Localization-Modifiability.



- An installation manual that is organized so that each user option and all installation-specific parameters are described in one place makes updating the manual easier when changes in these options and parameters occur.
- A cross-reference listing for a program localizes the description of how a particular variable is used, and thereby makes it easier to evaluate the effects of potential changes.

These examples can only serve to illustrate the style of approach (on a per-cube basis) used to make the goals, principles, and parts of the fundamental process useful in describing and understanding aspects of software engineering. The following sections discuss the hierarchy, goals, and principles in more specific terms before we apply them jointly in an extended example.

Hierarchical Decomposition

The process of developing hierarchical structure may be viewed *top-down* as successively imposing increasingly specific constraints on the form of an ultimate solution. It may also be viewed *bottom-up* as successively exploiting the constraints satisfied by lower levels. An understanding of software engineering lies in understanding the *constraints* each engineering activity places on the others—e.g., how the design constrains the implementation, and how the implementation possibilities constrain the design. The process of hierarchically and iteratively imposing constraints appropriate to the analysis, design, specification, and coding phases is what unifies the practice of software engineering.

The process of hierarchical decomposition involves both analysis and synthesis—both taking apart and putting together. While one decomposes a subject by recognizing the *different* sub-parts of which it is composed, one must also pause to examine the overall decomposition and look for *similarities* among the lower-level constituents with an eye toward recomposing collections of them into larger constructs. For example, pure decomposition would never result in recognizing subroutines that are needed in separated parts of the decomposition.

For a given problem there are many "correct" decompositions, and given a large collection of solutions to primitive sub-problems (e.g., a set of CPU instructions), there are many correct compositions which will solve a given problem. Thus, whether one is engaged in synthesis or analysis, composition or decomposition, the selection of a "correct" solution must be based upon some overriding criteria. We must try to select the most desirable solution from the set of correct solutions. For this reason we begin first, in our elaboration of the thesis of this paper, by considering the goals of software engineering.

The Goals of Software Engineering

The skill with which we can apply engineering methods and tools will depend on the degree to which we have a clear and precise view of our objectives. In the realm of software engineering our objectives will always be stated in terms of desired properties of the resultant software. Four properties that are sufficiently general to be accepted as *goals* for the entire discipline of software engineering are *modifiability*, *efficiency*, *reliability*, and *understandability*.

The goal of *modifiability* is historically the most difficult goal to master. Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained. The characterization of "sameness" or invariance may be very subtle, and the effects of change may be hard to predict. This makes the achievement of modifiability difficult. Modifiability is also difficult to achieve because changes occur for so many different types of reasons. For example, in transferring software to a new computer or operating system, it is desired to keep invariant the logical effects of the system, limiting changes only to the necessarily machine-dependent aspects. Changes are also required to remove errors from software, to add new capabilities, and to improve a system's performance. In general, different approaches are necessary to satisfy these different types of modifiability.

Modifiability requires not only the ability to have an adaptable, evolutionary design, employ standardized software building-blocks, tune for performance, etc., but also the more subtle ability to maintain project schedules and budgets by allowing dummy test modules to be used as drivers before later parts of a system are prepared, etc. The variety of ways modifiability affects software engineering is one of the reasons for giving it a primary role in our discussion of software engineering.

A much-abused goal is *efficiency*, usually because in an excess of zeal it is prematurely permitted a high priority in engineering tradeoffs. Blatant *inefficiency* cannot, of course, be tolerated, but usually efficiency questions are best treated within the context of other issues. For example, achieving a high degree of modifiability can provide the basis for meeting efficiency goals during the tuning phase of software development. In addition, insights

reflecting a more unified understanding of a problem have far more impact on efficiency (via abstraction and uniformity) than any amount of bit twiddling within a faulty structure. In general, except for the highest conceptual levels of a design, where gross inefficiency questions may play dominant roles, the efficiency goal does not dominate the practice of software engineering.

Reliability is a goal much in vogue today. Reliability must both prevent failure in conception, design, and construction, as well as recover from failure in operation or performance. Unlike efficiency, which frequently is prematurely applied, reliability is more often considered too late, or not at all, in most software development efforts. Reliability can only be built in from the start; it cannot be added on at the end. Hence, reliability has a pervasive and crucial effect on software engineering practices.

The final goal which should exert a strong influence in all aspects of software engineering is *understandability*. It depends, of course, on the intended audience: technical, management, or user. Note in particular that understandability is not merely a property of legibility. Much more importantly, the entire conceptual structure is involved. Also, in any given circumstance an acceptable level of understandability either is or is not present. There is no middle ground. Although understandability is, in a sense, a prerequisite to reliability and modifiability, it is also important as a goal in itself because it draws attention to an important barrier to understandability—complexity. Management of complexity is a crucial aspect of software engineering methods, and the need to manage complexity arises from the goal of understandability. The only way to achieve the goal of understandability with regard to an inherently complex system is to impose an appropriate structure and organization on the system. The structure must be represented in a clear notation that permits mechanical translators—e.g., compilers, etc.—to bridge the gap between the actual system and an understandable representation of it. Thus, achieving understandability depends as much upon the software engineering tools as on the methods. For example, when compilers do not produce acceptably efficient code, assembly language may have to be used, at a cost in program understandability.

The Principles of Software Engineering

The principles of software engineering are, as we mentioned earlier, *modularity*, *abstraction*, *localization*, *hiding*, *uniformity*, *completeness*, and *confirmability*. These *principles* applied in various combinations within the *fundamental process* will work to produce *hierarchical decompositions* which achieve our *goals* during all of the various phases of software development. The hierarchical decomposition of a system depicts the constituents of the system organized into a structure by the relationships among those constituents. The above seven principles, singly and in combination, are used to determine and control those relationships. They are used essentially as decision criteria to ensure that the resulting decomposition attains our goals, and thus each deals with some aspect of the relationships—i.e., the interfaces among the constituents.

Modularity Modularity deals with properties of hierarchical software structures. It has been given various definitions. Sometimes it has been defined in terms of objectives:

> "[One objective of modular programming is to be able to convince oneself] of the correctness of a program module, independently of the context of its use in building larger units of software." (Reference 2, p. 129)

> "Modularity denotes the ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules." (Reference 9, p. 54)

> "Modularity is not . . . simply the arbitrary division of a large program into smaller parts or modules. The primary goal . . . should be to decompose the program in such a way that the modules are highly independent from one another." (Reference 13, p. 100)

Modularity is more frequently defined in terms of structural properties possessed by "modular" systems:

> "A program is modular if it is written in many relatively independent parts or modules which have well-defined interfaces such that each module makes no assumptions about the operation of other modules except what is contained in the interface specifications." (Reference 4, p. 1)

> "Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. . . . A definition of "good" modularity must emphasize the requirement that modules be as disjoint as possible." (Reference 11, p. 192)

> "A modular program is a program [having a hierarchical structure]. (Reference 1, p. 34)

> "Modular programming is the organizing of a complete program into a number of small units . . . where there is a set of rules which controls the characteristics of those units. (Cited in Reference 10, p. 29)

In general, modularity is cited as helping to improve software reliability, helping to allow multiple use of common designs and programs, and helping to make it easier to modify programs. Most discussions of modularity focus on one or another of these objectives and attempt to explain why certain structural constraints make the attainment of these objectives easier.

Rather than select any one objective as the most important one, we propose[5] a general unifying definition:

> Modularity deals with how the *structure* of an object can make the attainment of some *purpose* easier. Modularity is *purposeful structuring*.

Hence, the principle of modularity is made concrete by explaining how certain constraints on the structure of systems can make it easier or harder to achieve some purpose.

For example, what sort of structural constraints facilitate modifiability? efficiency? reliability? Imposing such constraints on structures is the essence of applying the modularity principle in software engineering.[6,7] For example, goto-free programming forces programmers to make explicit the conditions under which a given statement is executed, and this can help ensure understandability and prevent errors.

The principle of modularity can be further illustrated by considering modularity issues arising in deciding what part/whole relationships should be considered in developing hierarchical decompositions:

Increasing Machine Dependence The lower the module in the system structure, the more the module is dependent on the hardware on which it runs. This helps to make software more portable, by isolating machine-dependencies.

Refinement of action Higher level modules specify objectives for some action, i.e., *what* is to be done. Lower levels describe how the objective is going to be realized, i.e., *how* something is to be done. For example, within a higher level module, an engineer might specify that a temperature is to be adjusted until it is at least 145°. A *lower level* module will define how this adjustment is performed, e.g., by adjusting and sampling the temperature trend at five minute intervals. This constraint helps make a system more understandable and easier to modify.

Scope of control Higher level modules call lower level modules and supervise their activities. This means that if lower level modules encounter some exception condition (e.g., a condition that prevents the requested operation from being performed), this must be reported to higher level modules so they can take appropriate action.

It may be impossible for a given program to satisfy all these objectives simultaneously. A program may have one structure if modules are ordered according to one rule, and a different structure if a different rule is considered. The main point in identifying these relationships is to highlight possible criteria that can be consciously used in structuring the modules. Many of these criteria are already used instinctively; the advantage of making the criteria explicit is that any programmer produces better results if he is consciously aware of criteria for evaluating what he is doing.

Abstraction Like modularity, *abstraction* is a very pervasive principle. The essence of abstraction is to extract essential properties while omitting inessential details. Our discussion of hierarchical decomposition in the form of "levels" showed abstraction in perhaps its most pristine form. Each level of the decomposition presents an abstract view of the lower levels purely in the sense that details are subordinated to the lower levels.

The principle of abstraction when combined with the principle of completeness ensures that a given level in a decomposition is understandable as a unit, without requiring either knowledge of lower levels of detail, or necessarily how it participates in the system as viewed from a higher level. Thus this principle is employed on the one hand to obtain a description of some level of the system which could be realized by any of several implementations, and on the other hand to give a description of one part of a system which could be used in many other systems requiring the same component at that level of abstraction.

The principle of abstraction interacts very strongly with the purpose underlying any particular decomposition. The principle is of little practical value unless combined with the principle of modularity ("purposeful structuring") to ensure that appropriate abstractions are found. Abstractions employed to achieve the goal of understandability mean that each level of abstraction, while presenting more and more detailed views of the system, must do so in terms which are understandable to the intended audience.

Localization The principle of *localization* is concerned with physical proximity. Things must be brought together all in one place. Thus, localization deals with physical interfaces, textual sequence, memory, etc. Then the other principles can interrelate the localized things to serve particular purposes. Consider carefully how intimate is the connection between localization of an abstraction in a module and our ability then to understand and deal with it.

Subroutines, arrays, logical and physical records, as well as paged memories, are examples of localization. The avoidance of goto's in structured programming is an application of localization to control structures which enhances understandability and simplifies confirmability.

Hiding Another principle familiar to many readers is *hiding*. Parnas,[13] for example, uses it as the major criterion for a decomposition into modules. It is related to the idea of "postponing binding decisions" in top-down problem-solving, although it is not the same. The purpose of hiding is similar to that of abstraction in that it requires making visible only those properties of a module needed to interface with other modules. But hiding differs from abstraction in that the purpose of hiding is *to make inaccessible* certain details that should not affect other parts of a system. Abstraction helps to identify details that should be hidden. Hiding is concerned with *defining and enforcing access constraints* that, without the hiding principle, would otherwise only be implicit in some purpose, concept, mechanism, notation, or usage description.

Hiding, combined with abstraction and localization, forces suppression of *how* to emphasize *what*. Suppressing how a constraint is satisfied forces the constraint to be made more explicit, thereby amplifying our understanding of the constraint itself. Deciding what constraints are to be expressed (an aspect of modularity—purposeful constraint selection) is a matter independent of the hiding principle itself.

Uniformity Uniformity (the lack of inconsistencies and unnecessary differences) is also an important principle. When applied to notational matters, uniformity yields a notation free of confusing and perhaps costly inconsistencies. When also combined with the abstraction principle, uniformity implies a notation that permits arbitrary mechanization of the internal detailing of a mechanism—the notation does not constrain one's choice of implementation. And when the hiding principle is added, the result is a notation that does not merely permit several implementation choices, but also ensures that no unnecessary details of a specific implementation are revealed by the notation. For example, if a subroutine parameter is to be a stack, the representation of the stack should usually be hidden from the user of the subroutine. Thus, the user should be able to allocate storage for stacks and pass them to subroutines without having access to the individual components of a stack. A notation for representing such abstract data types is given in References 7 and 12. Another example is given in a recent paper on exception handling methods,[8] in which a uniform notation is proposed whose semantics can be realized using various traditional methods of handling error returns from subroutines.

In its essence, notation satisfying the uniformity and abstraction concepts has been called elsewhere[16] the

Uniform Referent Principle. This principle, applied to the concept formation step of the fundamental process, yields a consistent and well-defined set of semantic concepts that can be represented with a uniform syntax. A uniform notational syntax is not possible if there is no semantic uniformity underlying the notation.

Completeness Completeness is obviously an important principle. The purpose of this principle is to ensure that all the essentials of an abstraction, for example, are explicit and that nothing essential has been omitted. This does not require that every *detail* be shown—merely that the set of abstract concepts *covers* every detail. Applied to notational matters, completeness requires that a notation provides a means for saying everything that one wants to say. Combined with abstraction, it implies that a notation should be concise, permitting the suppression of invariant details in favor of highlighting the potentially changeable. Completeness combined with uniformity and abstraction and applied to the goal of efficiency suggests that programmers should be able to select different implementation mechanisms to tune a system's performance, but without changing the form of any subroutine call, for example. A notation that does not permit this purpose to be achieved is incomplete.

Confirmability Confirmability is a principle that directs attention to methods for finding out whether stated goals have been achieved. Applied to design issues, confirmability refers to the structuring of a system so it is readily tested. It must be possible to stimulate the constructed system in a controlled manner so its response can be evaluated for correctness. Applied to notational matters, confirmability means that a notation should require explicit specification of constraints that affect the correctness of a design or implementation (e.g., data declarations that specify range of values and units of value as well as mode of representation). Applied to the practice of software engineering, confirmability refers to the use of such methods as structured walk-throughs of designs, egoless programming,[18] and other methods that help to ensure that nothing has been overlooked.

The principle of confirmability can be realized in many useful forms, both as entirely manual procedures and strongly aided by the tools and data base of a software engineering facility. Certain kinds of type checking and consistency checking reflect the principle of confirmability applied to the design of programming languages and compilers.

Completeness and confirmability are easily confused. For example, in the Introduction, we presented examples illustrating the interaction between notation, completeness, and reliability. In one of these examples, we noted that to ensure completeness of case statement control a programmer should be *permitted* by the syntax to specify what should happen when a case statement variable is out of range. Confirmability applied to the same issue would imply a programmer should be *required* to state what should happen. Of course, if he knows that out-of-range values are not possible, this too should be expressible, to permit implementation efficiency. In short, the evolution of completeness to satisfy confirmability requires that otherwise obscure implications be made to show in explicit form.

An Example of the Framework's Utility

Having discussed the components of the process/goal/principle framework at greater length, we now intend to show how the framework can be used to gain and structure insights into aspects of software engineering. By giving an extended example, we hope to show that the framework is not merely taxonomic but can actively assist in dealing with the complexities of software engineering.

Our example will show how the framework can help to organize our understanding of the notion of a subroutine. We choose this example because, although "subroutine" is fundamental to software, and seems well-understood, it is also one of the most complex concepts when considered in its totality. Figure 8 shows the pattern of the fundamental process applied to the subroutine concept. The notation proposed is essentially that of ALGOL 60. Other notations could have been proposed equally well. The description of the subroutine concept in Figure 8 is obviously very general, in particular, the description of "Mechanism" is at a high level of abstraction.

Applying hierarchical decomposition, the mechanism aspect of subroutines can be decomposed into two less-abstract mechanisms for implementing the subroutine concept—one for inline subroutines and one for closed subroutines. Then, the fundamental process can be applied again with respect to these mechanisms, as shown in Figures 9 and 10. We have inserted parenthetical comments to show how various principles and/or goals are being served.

Consider now the closed subroutine, Figure 9. Our description holds no surprises, for we are still at a high level. The notions of Figure 9 could be refined in several ways, however. For example, there are at least two distinct kinds of subroutine linkage mechanisms: 1) *direct linkage*, which is usually supported by a machine instruction that saves the return address and transfers control to the subroutine body, and 2) *indirect linkage*, a mechanism employed in AED implementations,[19] in which a subroutine call is implemented not directly by transferring control to the subroutine, but indirectly, by transferring control to a "linkage" subroutine. The purpose of this is to save space on machines for which stack manipulations are expensive and to *localize* at run-time all subroutine calls so different linkage routines can be substituted—e.g., a timing linkage that gathers information about how much time is spent in each subroutine, or a debugging linkage that permits interception and tracing of calls by a debugging package. This application of the localization principle to subroutine linkage has the advantage of making it easier to satisfy the efficiency goal (by gathering timing information important in tuning a system) and the reliability goal (by making it easier to track down bugs). Furthermore, when complex calling sequences are required by the language implementation, the slight run-time cost of enter and leave macro operations yields significant storage savings through localization and sharing of the machine instructions needed for each call.

The call-return *concept* could also be explicated further by discussing more specific examples of the concept, in the style of Figure 9. For example, the use of stack frames (e.g., see Reference 14) vs. the storing of return addresses and other information related to subroutine invocation in each subroutine's storage space can be clarified by

Purpose: To invoke a similar pattern of action from many places in a program.

Concept: "Similar" but not "the same" implies that the "same" parts must be collected in one place, but combined in each case with the "different" parts. The "combination" must show how the "different" and "same" parts interact.

Mechanism: The "same" part is a subroutine body, located in one place and referenced by name. The "different" parts are the actual parameter values associated in each case with the subroutine name. The parameter-passing mechanism and coding of the subroutine body implement the "combination" of the parameters with the body at call time.

Notation:

<procedure statement> ::= <procedure identifier> [([<actual parameter>{,<actual parameter>}])]

<actual parameter> ::= { <string> | <expression> | <array identifier> | <switch identifier> | <procedure identifier> }

<procedure declaration> ::= [<type>] procedure <identifier> [([<identifier>{,<identifier>}])];
→ {value <identifier>{,<identifier>};}
→ {<specifier> <identifier>{,<identifier>};}
→ <statement>

Usage: The syntax and semantics of a programming language define the contexts in which procedures can be invoked.

Figure 8. Top Level Explication of the Subroutine Call Concept

explicitly expressing purpose, concept, mechanism, notation, and usage.

It is useful to note the difference in the "Purpose" components of Figures 9 and 10. Although the goals inherited by decomposition from Figure 8 are the same—improve efficiency—note in Figure 10 that inline subroutines can improve efficiency in two ways: 1) by eliminating subroutine call overhead and 2) by performing certain computations at compile-time rather than at run-time, using actual parameter values. For example, if all parameters are constants, it may be possible to compute the value of the subroutine at compile time with consequent enormous savings in run-time efficiency. Wegbreit[18] explores this possibility and related ones in more detail.

For purposes of illustrating the use of our proposed framework, however, it is worth noting how Figures 9 and 10 help in comparing and contrasting two related but differing interpretations (inline versus closed) of the basic

Purpose:
- To save space by executing the same body of code with different parameter values

Concept:
- Call-return capability (transfer control to the subroutine's body, remembering where the call came from)

Mechanism:
- Specific calling sequences, e.g.
  - direct linkage
  - indirect linkage

Notation:
- Notation for call should not be different from notation used to invoke inline subroutines. (Note that this is an application of the uniformity principle, and serves to foster program modifiability.)

Usage:

Figure 9. Description of the Closed Subroutine Concept

Purpose:
- To save time by eliminating call overhead (the efficiency goal);
- To save execution time by permitting compile-time simplification of a subroutine body based on actual values of parameters

Concept:
- The subroutine body is substituted in place of the call, with actual parameter values substituted for formal parameter values

Mechanism:
- Macro substitution, followed by compile-time optimization; or
- Syntactic substitution, in the sense that local variables declared within the subroutine will not be found to conflict with similarly-named variables in the context of the call, as might happen with macro substitution (which occurs at the lexical level of a program text).

Notation:
- Should not be different from call notation used to invoke closed subroutines. (This is an application of the uniformity principle, and serves to foster program modifiability.)

Usage:

Figure 10. Description of the Inline Subroutine Concept

subroutine notion. Note also how we have applied the framework (using hierarchical decomposition) to alternative "Mechanism" and to alternative "Concept" components of the patterns in Figures 9 and 10. This recursive application of the pattern within components of the pattern is a principal attraction of using the framework for understanding software engineering topics in depth.

Our illustration so far appears to imply that the process aspect of the framework is of paramount importance, because we have organized our examples primarily in terms of this pattern. But each of the components of the framework is of equal importance, so an analysis can be organized equally well in terms of goals or principles. Depending on which dimension is used, the emphasis will be different, but using any of the three components as the dominating organizing concept is valid in applying the framework. Which of them should be used depends on the purpose of the discussion.

To demonstrate the validity and value of using one of the other components as the organizing principle, we describe in Figure 11 the notation aspect of subroutines, organized in terms of principles satisfied by various subroutine call notations. We will discuss each briefly below.

The purpose of *confirmability* as applied to notational matters is to ensure that important properties of a

Confirmability:

Purpose:	To ensure errors in forming a call are detectable.
Concept:	Distinguish input and output parameters
Mechanism:	Use a colon or a semicolon to separate input and output parameters, e.g., F(A,B,C,D) or F(A,B,C,D).
Concept:	Provide indication of what exceptions can be raised.
Mechanism:	See below, under Completeness.

Uniformity:

Purpose:	Avoid unnecessary differences in form of call.
Concept:	Ensure closed and inline calls have the same notational form. (This enhances modifiability directly, and efficiency indirectly.)

Completeness:

Purpose:	To ensure all properties of subroutines are reflected in the notation.
Concept:	Parameters should be both readable and writeable; notations for expressing response to exceptions should be available for use.
Mechanism:	• For read/write parameters: Use punctuation to separate input and output parameters Declare which parameters are input and which are output. Incorporate body of subroutine in program where it is referenced so global analysis can determine which parameters are input and which are output (e.g., as in ALGOL). • Notations to deal with the various types of exception conditions

Localization:

Purpose:	To express only once otherwise redundant information concerning exceptions.
Concept:	Associate handler with larger syntactic unit than the call itself.
Mechanism:	Use notation suggested in Reference B.

Hiding:

Purpose:	To prohibit access to information that should be available only to the subroutine.
Concept:	Use abstract data type in declaring an actual parameter's data type, but a more detailed declaration of the formal parameter's type.

Abstraction:

Purpose:	To preserve essential properties of call in the notation, leaving other details to other notational devices
Concept:	The method for handling exceptions should not be closely linked to implementation methods; a choice of a variety of implementation techniques should not be foreclosed by the notation.
Mechanism:	Use an implementation neutral notation for exception handling

Modularity:

Purpose:	To ensure that syntactic structure fosters appropriate goals.
Concept:	Examine impact of syntax on goal achievement.
Mechanism:	Choose the JOVIAL method of distinguishing input/output parameters rather than a declarative method, since the JOVIAL notation improves understandability.

Figure 11. Applying the Framework to Subroutine Call Notation Issues

subroutine's interface are stated explicitly in a call, so it is clear whether they have all been dealt with correctly. Two aspects of a call deserve particular mention as concepts for achieving this purpose. The first is to distinguish in the notation of the call which parameters are read-only (i.e., which are input parameters) and which are writeable (i.e., output parameters). The second is to distinguish in the form of the call what exception conditions a subroutine can raise.

PL/I is deficient in that no indication of output parameters is made. In this respect JOVIAL is superior, since a call to a JOVIAL subroutine, e.g., F(A,B,C,D), shows explicitly that A and B are input parameters and C and D are output parameters. In this case, the JOVIAL syntax is an example of a notational mechanism for implementing this concept. An equally good mechanism (considered solely from a confirmability viewpoint) would be merely to require that in the declaration of a subroutine, the input/output attributes of parameters be specified explicitly so the requirement can be checked at compile-time. Also, uniformity with more modern programming languages, as well as ordinary English, might say that the ":" of JOVIAL might better be a "," to further improve the understandability of its syntax.

The explicit indication of exception conditions is a confirmability issue in that it permits oversights with respect to exception conditions to be detected more easily. We will discuss this concept further under Completeness.

As for *uniformity*, we list merely the concept that inline and closed subroutine invocations should have the same form. This enhances modifiability for tuning (efficiency) purposes, since changing a decision about whether to treat a subroutine as closed or inline will not then require changing every call.

Under *completeness*, we again list the concept that a subroutine's invocation notation should provide for dealing with exception conditions. The purpose served here is not to ensure that all conditions are dealt with appropriately (as for confirmability) but rather to ensure that every capability of the subroutine concept is mapped into a suitable notation for invoking that capability. The ability to control the response to an exception is an important aspect of subroutine invocation, and notation should be provided to deal with it. The confirmability purpose perhaps provides a stronger argument for associating exception handling with calls, but we cite it here because it also satisfies the completeness principle as well. Similarly, introducing the ability to assign to parameters is a concept suggested by the completeness principle, distinguishing input and output parameters in the form of the call or in a declaration is an application of the confirmability principle, because this make it easier to detect errors at compile time.

The purpose of *localization* as applied to notational matters for dealing with exception conditions might be stated as, "When the same exception handler is to be associated with several calling points for the same subroutine, the notation for exception handling should permit the handler to be written once, rather than requiring it to be written as part of each call." One concept for satisfying this purpose is to associate handlers with larger syntactic units of text than the call itself e.g., with statements, loop bodies, etc. This proposal is explored further in Reference 8 and a specific syntax is proposed there.

The *hiding* principle applied to subroutine call notation means allowing the subroutine user access *only* to the abstract data type information needed for the semantics of the call. Thus declarations of formal parameters (i.e., on the inside of the subroutine) are broken into two parts, and only the abstract part is made available to the user for his declarations (e.g., see Reference 12).

The principle of *abstraction* combined with uniformity suggests that the notation for dealing with exception conditions should be *neutral* with respect to various implementation techniques for handling exceptions. In Reference 8, a notation for exception handling is proposed that can be implemented using status variables, return codes, subroutines passed as parameters, or PL/I ON conditions as implementation methods for dealing with exceptions, depending on the logical constraints associated with the exception. The point is that the notation should permit a programmer to deal with the abstract concept of an exception, without being tied down to implementation details until he is ready to tune a system. This point is explored in greater detail in the cited reference.

Finally, applying the *modularity* principle to notational matters means exploring how the structural constraints imposed by a syntax can help to achieve some purpose. For example, the goal of understandability is enhanced by JOVIAL's syntax for distinguishing input and output parameters, as opposed to declaring which parameters are input parameters but not distinguishing in the structure of the call which parameter is an input parameter. Of course, the JOVIAL notation degrades modifiability, in that should an input parameter ever be changed to an output parameter, all calls would have to be modified, whereas the localization inherent in a separate declaration of read/write properties would not have this drawback. Human judgement is used to make a tradeoff decision in this case. The point is that by considering the effect of syntactic structure on achieving some goal, we have shown how the modularity principle applies to notational issues.

In short, Figure 11 shows that it is equally possible to organize a discussion of aspects of the subroutine concept in terms of principles as in terms of the fundamental process/pattern. (As an exercise, the reader might consider what principles are satisfied or degraded by the notational concept of optional arguments.)

The analysis in Figures 8, 9, 10, and 11 is only the beginning of a complete explication of the subroutine concept, but we hope our discussion has shown that by recursively applying the framework, in conjunction with hierarchical decomposition, organized and insightful explications can be developed to account for the virtues and deficiencies of various software engineering purposes, concepts, mechanisms, notations, and usages.

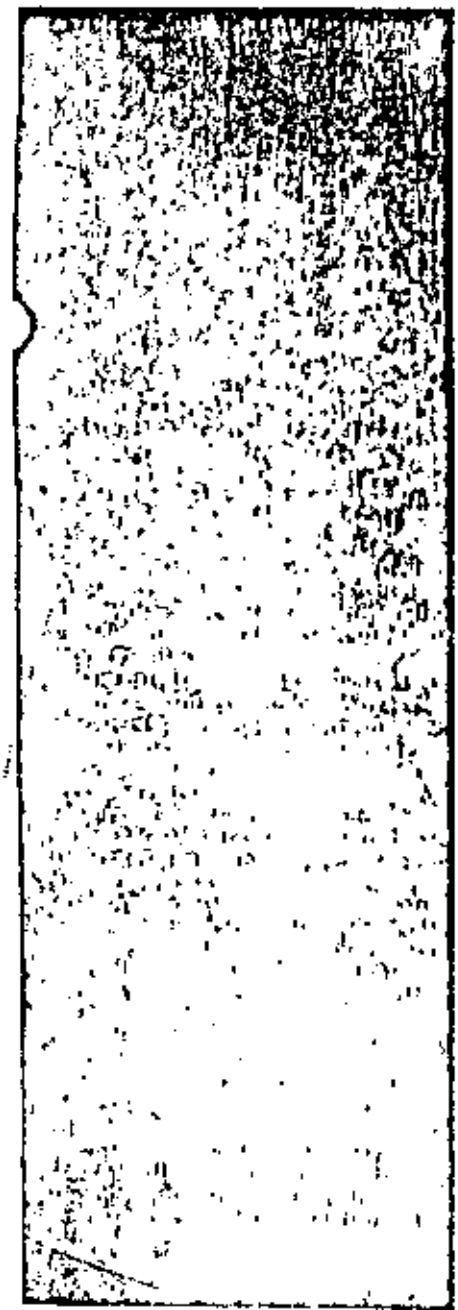
Conclusion

Our intent in this paper has been to consolidate and structure software engineering ideas into a coherent and useful framework for understanding the role these ideas play. The principles, goals, and process steps comprising this framework are not our inventions; they have been recognized by careful observers of software engineering for many years. We have merely attempted to present these ideas in an orderly and well-defined way. We do not claim to have identified all the important principles or goals

THE MYTHICAL MAN-MONTH

HOW DOES A PROJECT GET TO BE A YEAR LATE? . . . . . ONE DAY AT A TIME.

By Frederick P. Brooks, Jr.



NO SCENE FROM PREHISTORY is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and saber-toothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.

Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.

More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this case of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an unconscious assumption which is quite untrue, i.e., that all will go well.

Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable.

Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness required to make people wait for a good product.

Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.

Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline and thus begins a regenerative cycle which ends in disaster.

Schedule monitoring will be covered later. Let us now consider other aspects of the problem in more detail.

Optimism

All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of petty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: "This time it will surely run," or "I just found the last bug."

So the first false assumption that underlies the scheduling of systems programming is that *all will go well*, i.e., that *each task will take only as long as it "ought" to take*.

The persistence of optimism among programmers deserves more than a flip analysis. Dorothy Sayers, in her excellent book, *The Mind of the*

Maker, divides creative activity into three stages: the idea, the implementation, and the interaction. A book, then, or a computer, or a program comes into existence first as an ideal construct, built outside time and space but complete in the mind of the author. It is realized in time and space by pen, ink, and paper, or by wire, silicon, and ferrite. The creation is complete when someone reads the book, uses the computer or runs the program, thereby interacting with the mind of the maker.

This description, which Miss Sayers uses to illuminate not only human creative activity but also the Christian doctrine of the Trinity, will help us in our present task. For the human makers of things, the incompletenesses and inconsistencies of our ideas become clear only during implementation. Thus it is that writing, experimentation, "working out" are essential disciplines for the theoretician.

In many creative activities the medium of execution is intractable. Lumber splits; paints smear; electrical circuits ring. These physical limitations of the medium constrain the ideas that may be expressed, and they also create unexpected difficulties in the implementation.

Implementation, then, takes time and sweat both because of the physical media and because of the inadequacies of the underlying ideas. We tend to blame the physical media for most of our implementation difficulties; for the media are not "ours" in the way the ideas are, and our pride colors our judgment.

Computer programming, however, creates with an exceedingly tractable medium. The programmer builds from pure thought-stuff, concepts and very flexible representations thereof. Because the medium is tractable, we expect few difficulties in implementation; hence our pervasive optimism. Because our ideas are faulty, we have bugs; hence our optimism is unjustified.

In a single task, the assumption that all will go well has a probabilistic effect on the schedule. It might indeed go as planned, for there is a probability distribution for the delay that will be encountered, and "no delay" has a finite probability. A large programming effort, however, consists of many tasks, some chained end-to-end. The probability that each will go well becomes vanishingly small.

The mythical man-month

The second fallacious thought mode is expressed in the very unit of effort used in estimating and scheduling the man-month. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. *Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth.* It implies that men and months are interchangeable.

Men and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them* (Fig. 1). This is true of reaping wheat or picking cotton; it is not even approximately true of systems programming.

When a task cannot be partitioned

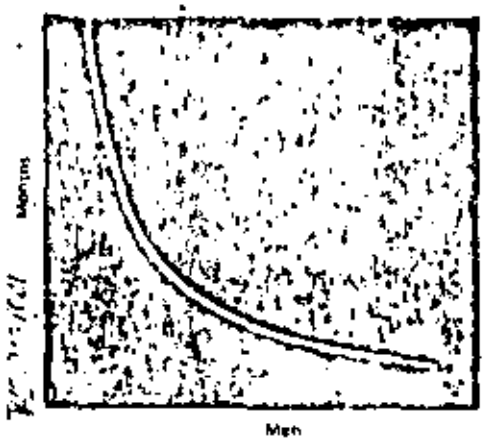


Fig. 1. The term "man-month" implies that if one man takes 10 months to do a job, 10 men can do it in one month. This may be true of picking cotton.

because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned. Many software tasks have this characteristic because of the sequential nature of debugging.

In tasks that can be partitioned but which require communication among the subtasks, the effort of communication must be added to the amount of work to be done. Therefore the best that can be done is somewhat poorer than an even trade of men for months (Fig. 2).

The added burden of communication is made up of two parts, training and intercommunication. Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies linearly with the number of workers.

V. A. Vyssotsky of Bell Telephone Laboratories estimates that a large project can sustain a manpower build-up of 30% per year. More than that strains and even inhibits the evolution of the essential informal structure and its communication pathways. F. J. Corbató of MIT points out that a long project must anticipate a turnover of 20% per year, and new people must be both technically trained and integrated into the formal structure.

Intercommunication is worse. If each part of the task must be separately coordinated with each other part, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two, four require six times as much as two. If, moreover, there need to be conferences among three, four, etc., workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task and bring us back to the situation of Fig. 3.

Since software construction is inherently a systems effort—an exercise in complex interrelationships—communication effort is great, and it quickly

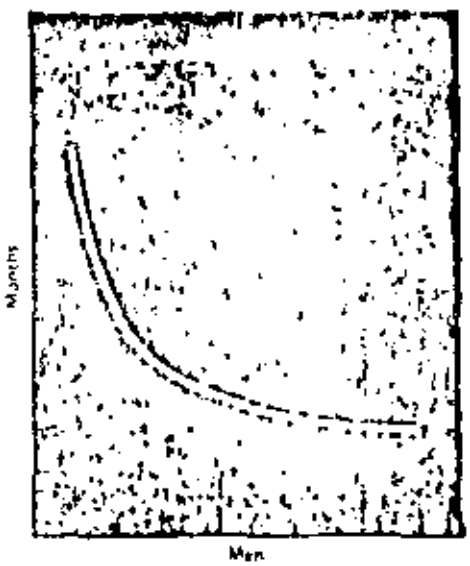


Fig. 2. Even on tasks that can be nicely partitioned among people, the additional communication required adds to the total work, increasing the schedule.

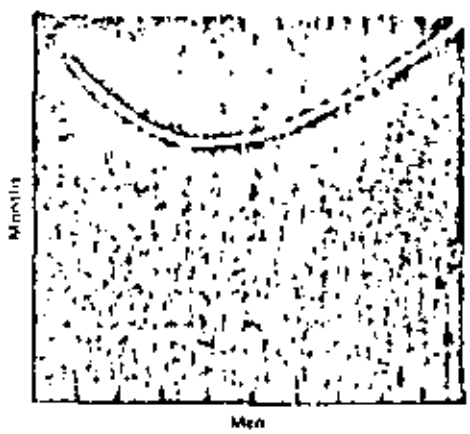


Fig. 3. Since software construction is complex, the communications overhead is great. Adding more men can lengthen, rather than shorten, the schedule.

dominates the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule.

Systems test

No parts of the schedule are so thoroughly affected by sequential constraints as component debugging and system test. Furthermore, the time required depends on the number and subtlety of the errors encountered. Theoretically this number should be zero. Because of optimism, we usually expect the number of bugs to be smaller than it turns out to be. Therefore testing is usually the most mis-scheduled part of programming.

For some years I have been successfully using the following rule of thumb for scheduling a software task:

- 1/3 planning
- 1/6 coding
- 1/4 component test and early system test
- 1/4 system test, all components in hand.

This differs from conventional scheduling in several important ways:

1. The fraction devoted to planning is larger than normal. Even so, it is barely enough to produce a detailed and solid specification, and not enough to include research or exploration of totally new techniques.
2. The *half* of the schedule devoted to debugging of completed code is much larger than normal.
3. The part that is easy to estimate, i.e., coding, is given only one-sixth of the schedule.

In examining conventionally scheduled projects, I have found that few allowed one-half of the projected schedule for testing, but that most did indeed spend half of the actual schedule for that purpose. Many of these were on schedule until and except in system testing.

Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date. Bad news, late and without warning, is unsettling to customers and to managers.

Furthermore, delay at this point has unusually severe financial, as well as psychological, repercussions. The project is fully staffed, and cost-per-day is maximum. More seriously, the software is to support other business effort (shipping of computers, operation of new facilities, etc.) and the secondary costs of delaying these are very high, for it is almost time for software shipment. Indeed, these secondary costs may far outweigh all others. It is therefore very important to allow enough system test time in the original schedule.

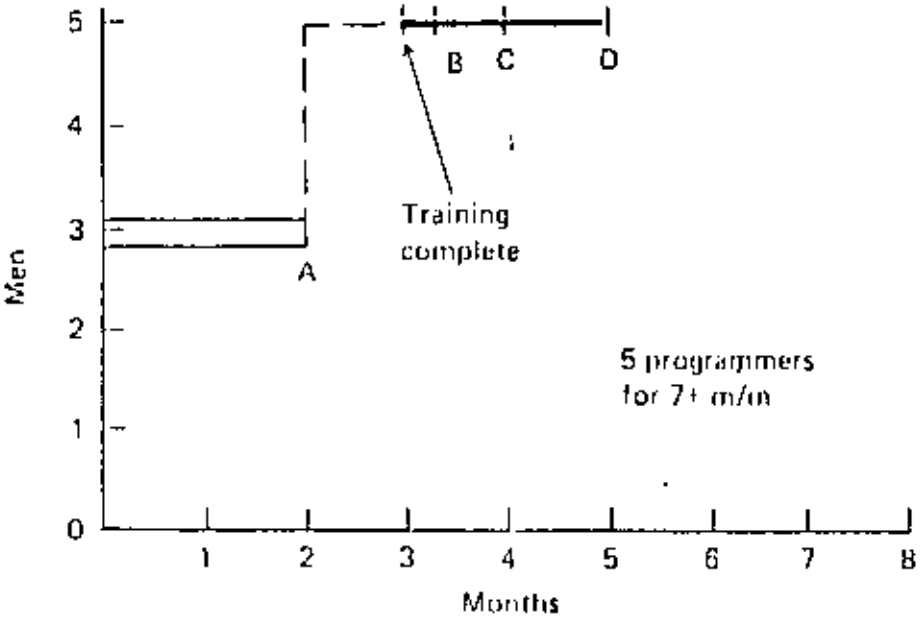


Fig. 4 Adding manpower to a project which is late may not help. In this case, suppose three men on a 12 man-month project were a month late. If it takes one of the three an extra month to train two new men, the project will be just as late as if no one was added.

Gutless estimating

Observe that for the programmer, as for the chef, the urgency of the patron may govern the scheduled completion of the task, but it cannot govern the actual completion. An omelette, promised in ten minutes, may appear to be progressing nicely. But when it has not set in ten minutes, the customer has two choices—wait or eat it raw. Software customers have had the same choices.

The cook has another choice; he can turn up the heat. The result is often an omelette nothing can save—burned in one part, raw in another.

Now I do not think software managers have less inherent courage and firmness than chefs, nor than other engineering managers. But false scheduling to match the patron's desired date is much more common in our discipline than elsewhere in engineering. It is very difficult to make a vigorous, plausible, and job-risking defense of an estimate that is derived by no quantitative method, supported by little data, and certified chiefly by the hunches of the managers.

Clearly two solutions are needed. We need to develop and publicize productivity figures, bug-incidence figures, estimating rules, and so on. The whole profession can only profit from sharing such data.

Until estimating is on a sounder basis, individual managers will need to stiffen their backbones, and defend their estimates with the assurance that their poor hunches are better than wish-derived estimates.

Regenerative disaster

What does one do when an essential software project is behind schedule? Add manpower, naturally. As Figs 1 through 3 suggest, this may or may not help.

Let us consider an example. Suppose a task is estimated at 12 man-months and assigned to three men for four months, and that there are measurable milepoints A, B, C, D, which are scheduled to fall at the end of each month.

Now suppose the first milepoint is not reached until two months have elapsed. What are the alternatives facing the manager?

1. Assume that the task must be done on time. Assume that only the first part of the task was misestimated. Then 9 man-months of effort remain, and two months, so 4½ men will be needed. Add 2 men to the 3 assigned.
2. Assume that the task must be done on time. Assume that the whole estimate was uniformly low. Then 18 man-months of effort remain, and two months, so 9 men will be needed. Add 6 men to the 3 assigned.
3. Reschedule. In this case I like the advice given by an experienced hardware engineer, "Take no small slips." That is, allow enough time in the new schedule to ensure that the work can be carefully and

thoroughly done, and that rescheduling will not have to be done again.

4. Trim the task. In practice this tends to happen anyway, once the team observes schedule slippage. Where the secondary costs of delay are very high, this is the only feasible action. The manager's only alternatives are to trim it formally and carefully, to reschedule, or to watch the task get silently trimmed by hasty design and incomplete testing.

In the first two cases, insisting that the unaltered task be completed in four months is disastrous. Consider the regenerative effects, for example, for the first alternative (Fig. 4 preceding page). The two new men, however competent and however quickly recruited, will require training in the task by one of the experienced men. If this takes a month, *3 man-months will have been devoted to work not in the original estimate.* Furthermore, the task, originally partitioned three ways, must be repartitioned into five parts, hence some work already done will be lost and system testing must be lengthened. So at the end of the third month, substantially more than 7 man-months of effort remain, and 5 trained people and one month are available. As Fig. 4 suggests, the product is just as late as if no one had been added.

To hope to get done in four months, considering only training time and not repartitioning and extra systems test, would require adding 4 men, not 2, at the end of the second month. To cover repartitioning and system test effects, one would have to add still other men. Now, however, one has at least a 7-man team, not a 3-man one; thus such aspects as team organization and task division are different in kind, not merely in degree.

Notice that by the end of the third month things look very black. The March 1 milestone has not been reached in spite of all the managerial effort. The temptation is very strong to repeat the cycle, adding yet more manpower. Therein lies madness.

The foregoing assumed that only the first milestone was miscalculated. If on March 1 one makes the conservative assumption that the whole schedule was optimistic one wants to add 6 men just to the original task. Calculation of the training, repartitioning, system testing effects is left as an exercise for the reader. Without a doubt, the regenerative disaster will yield a poorer product later, than would rescheduling with the original three men, unaugmented.

Oversimplifying outrageously, we state Brooks' Law:

Adding manpower to a late software project makes it later.

This then is the demythologizing of the man-month. The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined.

Calling the shot

How long will a system programming job take? How much effort will be required? How does one estimate?

I have earlier suggested ratios that seem to apply to planning time, coding, component test, and system test. First, one must say that one does *not* estimate the entire task by estimating the coding portion only and then applying the ratios. The coding is only one-sixth or so of the problem, and errors in its estimate or in the ratios could lead to ridiculous results.

Second, one must say that data for building isolated small programs are not applicable to programming systems products. For a program averaging about 3,200 words, for example, Sackman, Erikson, and Grant report an average code-plus-debug time of about 178 hours for a single programmer, a figure which would extrapolate to give an annual productivity of 35,800 statements per year. A program half that size took less than one-fourth as long, and extrapolated productivity is almost 80,000 statements per year.[10]. Planning, documentation, testing, system integration, and training times must be added. The linear extrapolation of such spring figures is meaningless. Extrapolation of times for the hundred-yard dash shows that a man can run a mile in under three minutes.

Before dismissing them, however, let us note that these numbers, although not for strictly comparable problems, suggest that effort goes as a power of size *even* when no communication is involved except that of a man with his memories.

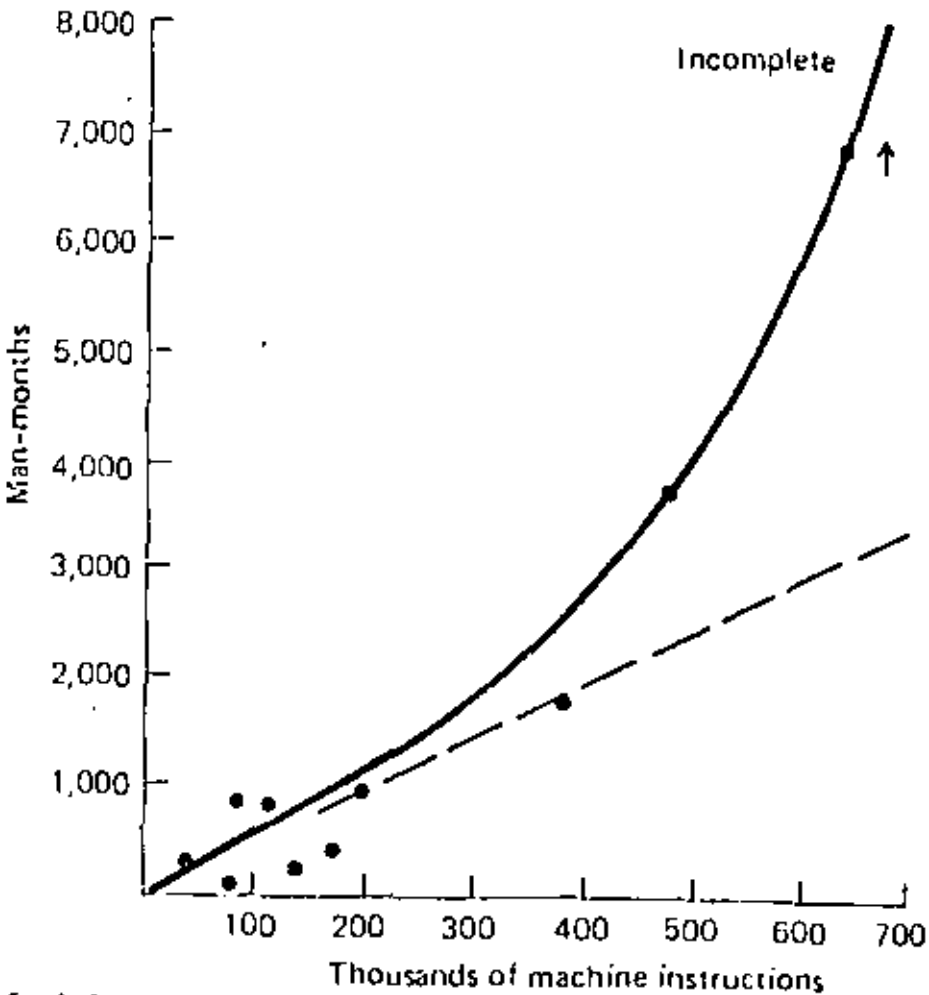


Fig. 5. As a project's complexity increases, the number of man-months required to complete it goes up exponentially.

Fig. 5 tells the sad story. It illustrates results reported from a study done by Nanus and Farr[5] at System Development Corp. This shows an exponent of 1.5, that is,

effort = (constant) × (number of instructions)$^{1.5}$

Another SDC study reported by Weinwurm[3] also shows an exponent near 1.5.

A few studies on programmer productivity have been made, and several estimating techniques have been proposed. Morin has prepared a survey of the published data.[4] Here I shall give only a few items that seem especially illuminating.

Portman's data

Charles Portman, manager of ICL's Software Div., Computer Equipment Organization (Northwest) at Manchester, offers another useful personal insight.

He found his programming teams missing schedules by about one-half—each job was taking approximately twice as long as estimated. The estimates were very careful, done by experienced teams estimating man-hours for several hundred subtasks on a PERT chart. When the slippage pattern appeared, he asked them to keep careful daily logs of time usage. These showed that the estimating error could be entirely accounted for by the fact that his teams were only realizing 50% of the working week as actual programming and debugging time. Machine downtime, higher-priority short unrelated jobs, meetings, paperwork, company business, sickness, personal time, etc accounted for the rest. In short, the estimates made an unrealistic assumption about the number of technical work hours per man-*year*. My own experience quite confirms his conclusion.

An unpublished 1964 study by E. F. Bardain shows programmers realizing only 27% productive time.[4]

	Prog units	Number of programmers	Years	Man-years	Program words	Words/ man-yr.
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2¼	17	38,000	2230
Translator (Data assembler)	15	13	2½	11	25,000	2270

Table 1. Data from Bell Labs indicates productivity differences between complex problems (the first two are basically control programs with many modules) and less complex ones. No one is certain how much of the difference is due to complexity, how much to the number of people involved.

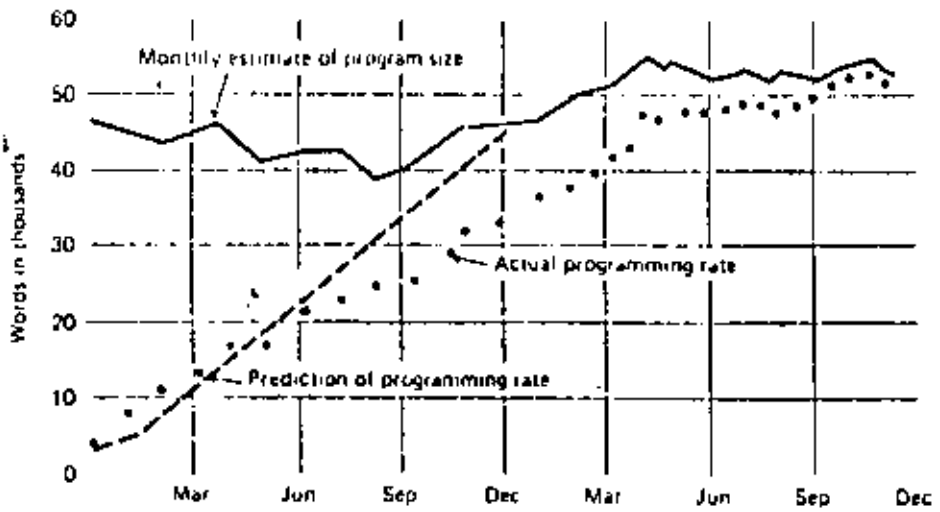


Fig. 6 Bell Labs' experience in predicting programming effort on one project.

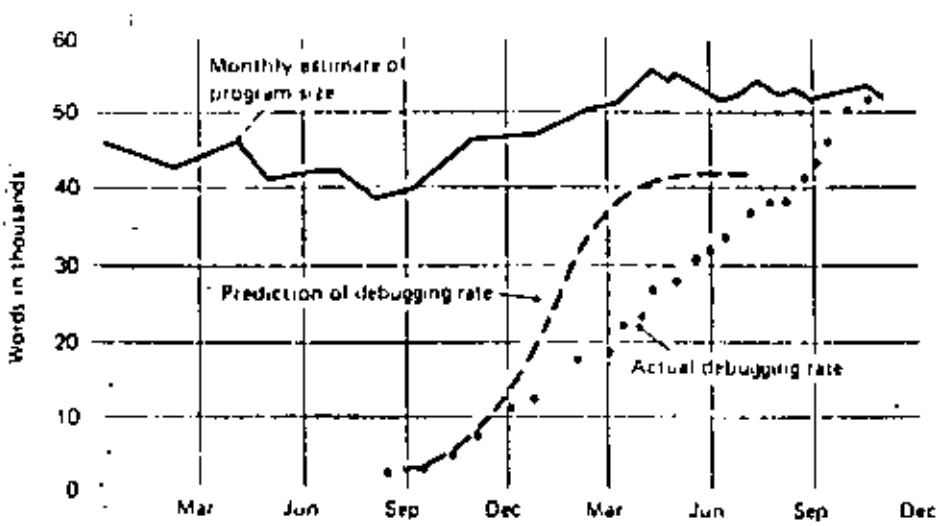


Fig. 7. Bell's predictions for debugging rates on a single project, contrasted with actual figures.

Aron's data

Joel Aron, manager of Systems Technology at IBM in Gaithersburg, Maryland, has studied programmer productivity when working on nine large systems (briefly, *large* means more than 25 programmers and 30,000 deliverable instructions). He divides such systems according to interactions among programmers (and system parts) and finds productivities as follows:

Very few interactions	10,000 instructions per man-year
Some interactions	5,000
Many interactions	1,500

The man-years do not include support and system test activities, only design and programming. When these figures are diluted by a factor of two to cover system test, they closely match Harr's data.

Harr's data

John Harr, manager of programming for the Bell Telephone Laboratories' Electronic Switching System, reported his and others' experience in a paper at the 1969 Spring Joint Computer Conference.[6] These data are shown in Table 1 and Figs. 6 and 7.

Of these, Fig. 6 is the most detailed and the most useful. The first two jobs are basically control programs, the second two are basically language translators. Productivity is stated in terms of debugged words per man-year. This includes programming, component test, and system test. It is not clear how much of the planning effort, or effort in machine support, writing, and the

like, is included.

The productivities likewise fall into two classifications: those for control programs are about 600 words per man-year; those for translators are about 2,200 words per man-year. Note that all four programs are of similar size—the variation is in size of the work groups, length of time, and number of modules. Which is cause and which is effect? Did the control programs require more people because they were more complicated? Or did they require more modules and more man-months because they were assigned more people? Did they take longer because of the greater complexity, or because more people were assigned? One can't be sure. The control programs were surely more complex. These uncertainties aside, the numbers describe the real productivities achieved on a large system, using present-day programming techniques. As such they are a real contribution.

Figs. 6 and 7 show some interesting data on programming and debugging rates as compared to predicted rates.

OS/360 data

IBM OS/360 experience, while not available in the detail of Harr's data, confirms it. Productivities in range of 600-800 debugged instructions per man-year were experienced by control program groups. Productivities in the 2,000-3,000 debugged instructions per man-year were achieved by language translator groups. These include planning done by the group, coding component test, system test, and some support activities. They are comparable to Harr's data, so far as I can tell.

Aron's data, Harr's data, and the OS/360 data all confirm striking differences in productivity related to the complexity and difficulty of the task itself. My guideline in the morass of estimating complexity is that compilers are three times as bad as normal batch application programs, and operating systems are three times as bad as compilers.

Corbató's data

Both Harr's data and OS/360 data are for assembly language programming. Little data seem to have been published on system programming productivity using higher-level languages. Corbató of MIT's Project MAC reports, however, a mean productivity of 1,200 lines of debugged PL/1 statements per man-year on the MULTICS system (between 1 and 2 million words).[7]

This number is very exciting. Like the other projects, MULTICS includes control programs and language translators. Like the others, it is producing a system programming product, tested and documented. The data seem to be comparable in terms of kind of effort included. And the productivity number is a good average between the control program and translator productivities of other projects.

But Corbató's number is *lines* per man-year, not *words!* Each statement in his system corresponds to about three-to-five words of handwritten code! This suggests two important conclusions:

- Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.
- Programming productivity may be increased as much as five times when a suitable high-level language is used. To back up these conclusions, W. M. Taliaffero also reports a constant productivity of 2,400 statements/year in Assembler, FORTRAN, and COBOL.[8] E. A. Nelson has shown a 3-to-1 productivity improvement for high-level language, although his standard deviations are wide.[9]

Hatching a catastrophe

When one hears of disastrous schedule slippage in a project, he imagines that a series of major calamities must have befallen it. Usually, however, the disaster is due to termites, not tornadoes; and the schedule has slipped imperceptibly but inexorably. Indeed, major calamities are easier to handle; one responds with major force, radical reorganization, the invention of new approaches. The whole team rises to the occasion.

But the day-by-day slippage is harder to recognize, harder to prevent, harder to make up. Yesterday a key man was sick, and a meeting couldn't be held. Today the machines are all down, because lightning struck the building's power transformer. Tomorrow the disc routines won't start testing, because the first disc is a week late from the factory. Snow, jury duty, family problems, emergency meetings with customers, executive audits—the list goes on and on. Each one only postpones some activity by a half-day or a day. And the schedule slips, one day at a time.

How does one control a big project on a tight schedule? The first step is to *have* a schedule. Each of a list of events, called milestones, has a date. Picking the dates is an estimating problem, discussed already and crucially dependent on experience.

For picking the milestones there is only one relevant rule. Milestones must be concrete, specific, measurable events, defined with knife-edge sharpness. Coding, for a counterexample, is "90% finished" for half of the total coding time. Debugging is "99% complete" most of the time. "Planning complete" is an event one can proclaim almost at will.[10]

Concrete milestones, on the other hand, are 100% events. "Specifications signed by architects and implementers," "source coding 100% complete, keypunched, entered into disc library," "debugged version passes all test cases." These concrete milestones demark the vague phases of planning, coding, debugging.

It is more important that milestones be sharp-edged and unambiguous than that they be easily verifiable by the boss. Rarely will a man lie about mile-

> None love
> the bearer of bad news.
> *Sophocles*

stone progress, *if* the milestone is so sharp that he can't deceive himself. But if the milestone is fuzzy, the boss often understands a different report from that which the man gives. To supplement Sophocles, no one enjoys bearing bad news, either, so it gets softened without any real intent to deceive.

Two interesting studies of estimating behavior by government contractors on large-scale development projects show that:

1. Estimates of the length of an activity made and revised carefully every two weeks before the activity starts do not significantly change as the start time draws near, no matter how wrong they ultimately turn out to be.
2. *During* the activity, *over*estimates of duration come steadily down as the activity proceeds.
3. *Underestimates* do not change significantly during the activity until about three weeks before the scheduled completion.[11]

Sharp milestones are in fact a service to the team, and one they can properly expect from a manager. The fuzzy milestone is the harder burden to live with. It is in fact a millstone that grinds down morale, for it deceives one about lost time until it is irremediable. And chronic schedule slippage is a morale-killer.

"The other piece is late"

A schedule slips a day; so what? Who gets excited about a one-day slip? We can make it up later. And the other piece ours fits into is late anyway.

A baseball manager recognizes a nonphysical talent, *hustle*, as an essential gift of great players and great teams. It is the characteristic of *running faster than necessary, moving sooner than necessary, trying harder than necessary*. It is essential for great programming teams, too. Hustle provides the cushion, the reserve capacity, that enables a team to cope with routine mishaps, to anticipate and forfend minor calamities. The calculated response, the measured effort, are the wet blankets that dampen hustle. As we have seen, one *must* get excited about a one-day slip. Such are the elements of catastrophe.

But not all one-day slips are equally disastrous. So some calculation of *response* is necessary, though hustle be dampened. How does one tell which slips matter? There is no substitute for a PERT chart or a critical-path schedule. Such a network shows who waits for what. It shows who is on the critical path, where any slip moves the end date. It also shows how much an activity can slip before it moves into the critical path.

The PERT technique, strictly speaking, is an elaboration of critical-path scheduling in which one estimates three times for every event, times corresponding to different probabilities of meeting the estimated dates. I do not find this refinement to be worth the extra effort, but for brevity I will call any critical path network a PERT chart.

The preparation of a PERT chart is the most valuable part of its use. Laying out the network, identifying the dependencies, and estimating the legs all force a great deal of very specific planning very early in a project. The first chart is always terrible, and one invents and invents in making the second one.

As the project proceeds, the PERT chart provides the answer to the demoralizing excuse, "The other piece is late anyhow." It shows how hustle is needed to keep one's own part off the critical path, and it suggests ways to make up the lost time in the other part.

Under the rug

When a first-line manager sees his small team slipping behind, he is rarely inclined to run to the boss with this woe. The team might be able to make it up, or he should be able to invent or reorganize to solve the problem. Then why worry the boss with it? So far, so good. Solving such problems is exactly what the first-line manager is "there" for. And the boss does have enough real worries demanding his action that he doesn't seek others. So all the dirt gets swept under the rug.

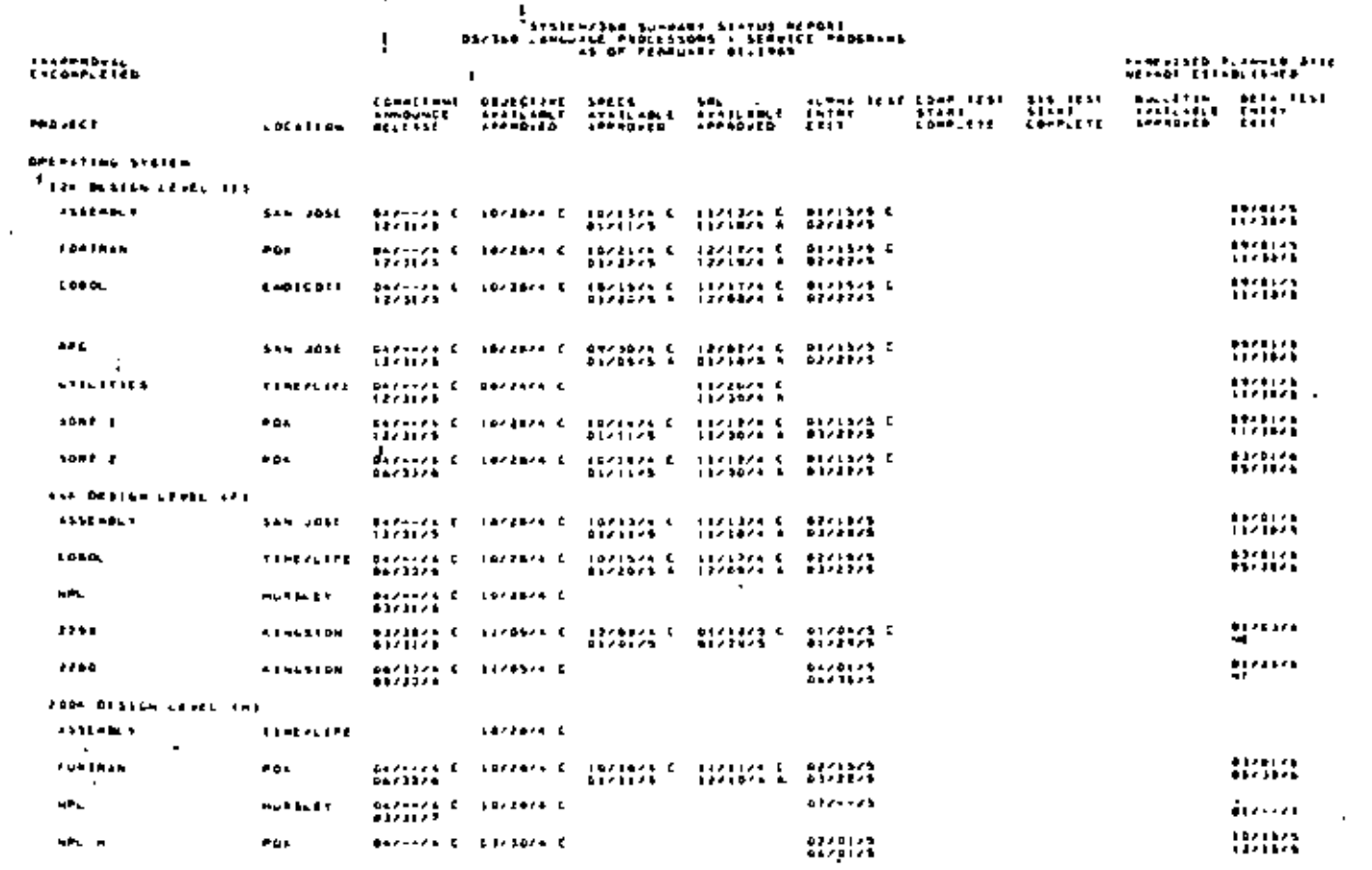
But every boss needs two kinds of information, exceptions for action and a status picture for education. For that purpose he needs to know the status of all his teams. Getting a true picture of that status is hard.

The first-line manager's interests and those of the boss have an inherent conflict here. The first-line manager fears that if he reports his problem, the boss will act on it. Then his action will preempt the manager's function, diminish his authority, foul up his other plans. So as long as the manager thinks he can solve it alone, he doesn't tell the boss.

Two rug-lifting techniques are open to the boss. Both must be used. The first is to reduce the role conflict and inspire sharing of status. The other is to yank the rug back.

Reducing the role conflict

The boss must first distinguish between action information and status information. He must discipline himself *not* to act on problems his managers can solve, and *never* to act on problems when he is explicitly reviewing status. I once knew a boss who invariably picked up the phone to give orders before the end of the first para-



SYSTEM/360 SUMMARY STATUS REPORT
DISTRIBUTED LANGUAGE PROCESSORS & SERVICE PROGRAMS
AS OF FEBRUARY 01,1969

PROJECT	LOCATION	COMMITMENT ANNOUNCE RELEASE	OBJECTIVE AVAILABLE APPROVED	SPECS AVAILABLE APPROVED	SRL AVAILABLE APPROVED	ALPHA TEST ENTRY EXIT	BETA TEST START COMPLETE	SYS TEST START COMPLETE	PUBLICATION AVAILABLE APPROVED	BETA TEST ENTRY EXIT
OPERATING SYSTEM										
360 DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/68 C 12/31/69	10/28/68 C	10/15/68 C 01/31/69	11/13/68 C 11/18/68 A	01/15/69 C 02/28/69				09/01/69 11/30/69
FORTRAN	POK	04/--/68 C 12/31/69	10/28/68 C	10/21/68 C 01/31/69	12/17/68 C 12/18/68 A	01/15/69 C 02/28/69				09/01/69 11/30/69
COBOL	ENDICOTT	04/--/68 C 12/31/69	10/28/68 C	10/15/68 C 01/30/69 A	11/11/68 C 12/06/68 A	01/15/69 C 02/27/69				09/01/69 11/30/69
APL	SAN JOSE	04/--/68 C 12/31/69	10/28/68 C	09/30/68 C 01/06/69 A	12/02/68 C 01/10/69 A	01/15/69 C 02/27/69				09/01/69 11/30/69
UTILITIES	KINGSTON	04/--/68 C 12/31/69	08/23/68 C		11/20/68 C 11/30/68 A					09/01/69 11/30/69
SORT 1	POK	04/--/68 C 12/31/69	10/28/68 C	10/15/68 C 01/11/69	11/13/68 C 11/30/68 A	01/15/69 C 01/27/69				09/01/69 11/30/69
SORT 2	POK	04/--/68 C 08/31/69	10/28/68 C	10/15/68 C 01/11/69	11/13/68 C 11/30/68 A	01/15/69 C 01/27/69				03/01/69 05/30/69
44K DESIGN LEVEL (F)										
ASSEMBLY	SAN JOSE	04/--/68 C 12/31/69	10/28/68 C	10/13/68 C 01/31/69	11/13/68 C 11/18/68 A	02/15/69 03/28/69				03/01/69 11/30/69
COBOL	KINGSTON	04/--/68 C 08/31/69	10/28/68 C	10/15/68 C 01/20/69 A	11/13/68 C 12/06/68 A	02/15/69 03/27/69				03/01/69 05/30/69
MPL	HURSLEY	04/--/68 C 03/31/69	10/28/68 C							
2250	KINGSTON	03/28/69 C 03/31/69	11/06/68 C	12/02/68 C 01/01/69	01/13/69 C 01/20/69	01/20/69 C 01/29/69				01/31/69 ND
2260	KINGSTON	06/13/69 C 08/31/69	11/05/68 C			04/01/69 04/15/69				01/31/69 ND
2004 DESIGN LEVEL (H)										
ASSEMBLY	KINGSTON		10/28/68 C							
FORTRAN	POK	04/--/68 C 08/31/69	10/28/68 C	10/18/68 C 01/31/69	11/11/68 C 12/10/68 A	02/15/69 03/28/69				03/01/69 05/30/69
MPL	HURSLEY	04/--/68 C 03/31/69	10/28/68 C			01/--/69				01/--/69
MPL H	POK	04/--/68 C	11/30/68 C			02/01/69 04/01/69				10/31/69 12/31/69

Fig 8. A report showing milestones and status is a key document in project control. This one shows some problems in OS development: specifications approval is late on some items (those without "A"); documentation (SRL) approval is overdue on another; and one (2250 support) is late coming out of alpha test.

graph in a status report. That response is guaranteed to squelch full disclosure.

Conversely, when the manager knows his boss will accept status reports without panic or preemption, he comes to give honest appraisals.

This whole process is helped if the boss labels meetings, reviews, conferences, as *status-review* meetings versus *problem-action* meetings, and controls himself accordingly. Obviously one may call a problem-action meeting as a consequence of a status meeting, if he believes a problem is out of hand. But at least everybody knows what the score is, and the boss thinks twice before grabbing the ball.

Yanking the rug off

Nevertheless, it is necessary to have review techniques by which the true status is made known, whether cooperatively or not. The PERT chart with its frequent sharp milestones is the basis for such review. On a large project one may want to review some part of it each week, making the rounds once a month or so.

A report showing milestones and actual completions is the key document. Fig. 8 (preceding page), shows an excerpt from such a report. This report shows some troubles. Specifications approval is overdue on several components. Manual (SRL) approval is overdue on another, and one is late getting out of the first state (ALPHA) of the independently conducted product test. So such a report serves as an agenda for the meeting of 1 February. Everyone knows the questions, and the component manager should be prepared to explain why it's late, when it will be finished, what steps he's taking, and what help, if any, he needs from the boss or collateral groups.

V. Vyssotsky of Bell Telephone Laboratories adds the following observation:

I have found it handy to carry both "scheduled" and "estimated" dates in the milestone report. The scheduled dates are the property of the project manager and represent a consistent work plan for the project as a whole, and one which is a priori *a reasonable plan. The estimated dates are the property of the lowest level manager who has cognizance over the piece of work in question, and represents his best judgment as to when it will actually happen, given the resources he has available and when he received (or has commitments for delivery of) his prerequisite inputs. The project manager has to keep his fingers off the estimated dates, and put the emphasis on getting accurate, unbiased estimates rather than palatable optimistic estimates or self-protective conservative ones. Once this is clearly established in everyone's mind, the project manager can see quite a ways into the future where he is going to be in trouble if he doesn't do something.*

The preparation of the PERT chart is a function of the boss and the managers reporting to him. Its updating, revision, and reporting requires the attention of a small (one-to-three-man) staff group which serves as an extension of the boss. Such a "Plans and Controls" team is invaluable for a large project. It has no authority except to ask all the line managers when they will have set or changed milestones, and whether milestones have been met. Since the Plans and Controls group handles all the paperwork, the burden on the line managers is reduced to the essentials—making the decisions.

We had a skilled, enthusiastic, and diplomatic Plans and Controls group on the OS/360 project, run by A. M. Pietrasanta, who devoted considerable inventive talent to devising effective but unobtrusive control methods. As a result, I found his group to be widely respected and more than tolerated. For a group whose role is inherently that of an irritant, this is quite an accomplishment.

The investment of a modest amount of skilled effort in a Plans and Controls function is very rewarding. It makes far more difference in project accomplishment than if these people worked directly on building the product programs. For the Plans and Controls group is the watchdog who renders the imperceptible delays visible and who points up the critical elements. It is the early warning system against losing a year, one day at a time.

Epilogue

The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks. The management of this complex craft will demand our best use of new languages and systems, our best adaptation of proven engineering management methods, liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations.

References

1. Sackman, H., W. J. Erikson, and E. E. Grant, "Exploratory Experimentation Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, 11 (1968), 3-11.
2. Nanus, B., and L. Farr, "Some Cost Contributors to Large-Scale Programs," *AFIPS Proceedings, SJCC*, 25 (1964), 239-248.
3. Weinwurm, G. F., *Research in the Management of Computer Programming*, Report SP-2059, 1965, System Development Corp., Santa Monica.
4. Migan, L. R., *Estimation of Resources for Computer Programming Projects*, M.S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Quoted by D. B. Mayer and A. W. Stalnaker, "Selection and Evaluation of Computer Personnel," *Proceedings 23rd ACM Conference*, 1968, 661.
6. Paper given at a panel session and not included in the *AFIPS Proceedings*.
7. Corbato, F. J., *Sensitive Issues in the Design of Multi-User Systems*. Lecture at the opening of the Honeywell EDP Technology Center, 1968.
8. Taliaferro, W., M., "Modularity the Key to System Growth Potential," *Software*, 1 (1971), 245-257.
9. Nelson, E. A., *Management Handbook for the Estimation of Computer Programming Costs*, Report TM-3225, System Development Corp., Santa Monica, pp. 46-67.
10. Reynolds, C. H., "What's Wrong with Computer Programming Management?" in *On the Management of Computer Programming*, Ed. G. F. Weinwurm. Philadelphia: Auerbach, 1971, pp. 35-42.
11. King, W. R., and T. A. Wilson, "Subjective Time Estimates in Critical Path Planning—a Preliminary Analysis," *Management Science*, 13 (1967), 307-320, and sequel, W. R. King, D. M. Witterrongel, and K. D. Hezel, "On the Analysis of Critical Path Time Estimating Behavior," *Management Sciences*, 14 (1967), 79-84.
12. Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition*. New York: Wiley, 1969, pp. 428-430. □

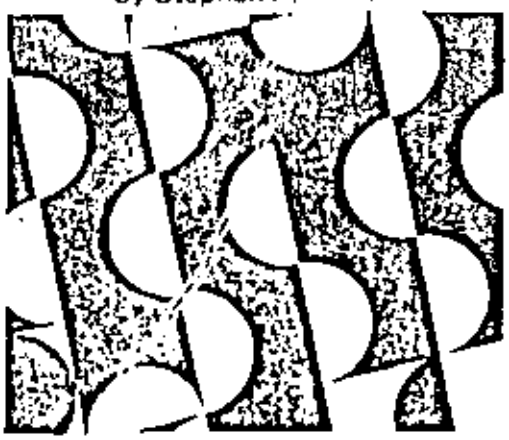
Dr. Brooks is presently a professor at the Univ. of North Carolina at Chapel Hill, and chairman of the computer science department there. He is best known as "the father of the IBM System/360," having served as project manager for the hardware development and as manager of the Operating System/360 project during its design phase. Earlier he was an architect of the IBM Stretch and Harvest computers.

At Chapel Hill he has participated in establishing and guiding the Triangle Universities Computation Center and the North Carolina Educational Computing Service. He is the author of two editions of "Automatic Data Processing" and "The Mythical Man-Month: Essays on Software Engineering" (Addison-Wesley), from which this excerpt is taken.

In contrast to Dr. Brooks' presentation, this portrait of failure is for those who learn best from looking at bad examples.

WHY PROJECTS FAIL

by Stephen P. Kaider



ONE OF THE PRIMARY causes for the failure of data processing projects is that such projects are often not initially defined, and therefore may lack a beginning and an end. Once a project has begun, no one seems to know:

- how the project was started;
- what the staffing is, or was, at any one point in time;
- what activities have been performed;
- when the project will end;
- what the project will accomplish.

Essentially, because projects are rarely formally defined, they are rarely completed. Completion occurs usually upon the death—or resignation—of the user the project services, or when the system is due for conversion. Completion is also a prerequisite for success, but a project is considered successful only if completed within the original time or budget estimates, and by how well it satisfies the user's needs.

An unsuccessful project, however, can be identified during several phases of its life cycle; and I shall here try to point to those very indicators.

Logically, any project can be time-divided into five distinct phases:

a) Pre-initiation period (usually measured in weeks or months)
b) Initiation period (measured in weeks)
c) Project duration (in months or years)
d) Project termination period (in weeks or months)
e) Post-termination period (occurring several months after project termination)

In each of the above phases, errors of commission or omission can have major impact upon the success of the total project.

Pre-initiation period

1) No standards exist for estimating how long the project will take. That is, each project is treated as a new and novel system with some individual responsible for estimation. His estimate will be based upon his own understanding of the project and its tasks, and on how quickly *he* can accomplish the subtasks. Little use is made of a history file of similar projects and actual versus originally estimated times.

2) Estimation is not done by the probable project leader, but rather, by whoever happens to be available at estimating time.

3) The project is not adequately defined. The request for an estimate usually takes the form of "John, we're planning to redo the payroll system. What do you think it will require?" "Payroll" may mean a number of different things to different people. Does it involve labor distribution? personnel information? leave accounting? salary, hourly and executive payroll? Any of the above can measurably impact the estimate of the project.

4) Short lead times are allowed for estimates, with corresponding inaccuracy as a result.

5) Personnel availability for the project is unknown. Estimates are usually prepared irrespective of who will perform the work. That is, an estimate of 34 days may be made, but only very junior personnel may be available; this will inflate the actual time. Although the resulting price/performance ratio may be excellent, the success of the project is rated in terms of actual versus estimated time, and on that basis the project may be a failure.

6) Staff desires are unknown. A project may be very appealing to one staff member, but repugnant to another. In both cases the actual time will be affected. Consequently, the Systems Manager must understand staff desires and assign projects accordingly where possible.

Initiation of project

1) Little documentation is available for existing, similar, or interfacing systems to provide the project leader with a data base to build upon.

2) Project leader responsibility is undefined. The leader has no idea what is expected of him, in regard to the project or the personnel assigned to work on it. Should he recommend alternative solutions? Can he recommend terminating the project? Can he remove personnel from it? Can he recommend dismissal?

3) Paper flow is handled poorly (or is nonexistent). Documentation regarding responsibilities, acceptance criteria, system objectives, etc., is not developed. Rather, documentation is limited to the technical aspects of the project.

4) Knowledge of "tools" to perform the project more efficiently is lacking. Are there modules, or subroutines already available which can be used? Is there a test data generator available? What about system design or documentation aids?

5) Definition of the project is vague, misleading, or totally wrong.

6) The project, between the time of the original estimate and its initiation, has changed without a corresponding change in the estimate.

7) Little or no time is spent in planning the project. Rather, analysis design and/or coding is begun immediately upon the project approval. The project leader is not permitted the "luxury" of planning: how he will attack the project; what tasks will be done first, second or third; what approach he will use; or what similar projects he will investigate or review.

8) Problem avoidance is not understood or considered. Oddly

enough, all projects begin with the premise that everything will go smoothly. Items such as lack of test time due to year-end closing are not considered until after the problem has occurred. By then, the project has already lost several days, or it is too late to provide an alternate source.

9) Resource requirements are not scheduled for the project. Critical items, such as keypunch, test time, user manual typing, secretarial, and printing requirements become a problem, and are addressed only *after* they have affected the project.

10) The project team's activities are not clearly presented to the end user. Only too often, the result is a series of "I thought . . ." "I assumed . . .? "Isn't he . . .?" comments.

11) Project completion elements are not defined. That is, the project leader is not aware of what constitutes completion of the project. What is the end product? What test/acceptance criteria will be used? Who must sign off on project turnover? What constitutes turnover?

Duration of the project

1) Posting or reporting of project information is not performed, resulting in the project leader being unaware of what the completion percentage is, and the user being unaware of the impact of changes upon the original system.

2) Project reviews are typically exercises in trivia. They constitute a "How's it going, Jack? Any problems? No? Good! See you next week." The weak systems manager does not ask probing, detailed questions. He does not require that his personnel anticipate problems, but is primarily concerned with identifying problems which his project leader already has recognized.

3) Change of personnel is one of the major reasons why projects fail. Personnel, including project leaders, are removed from the project, with no adjustments to the schedule for time lost due to the changes. Whenever a team member is added to a project, there is a learning curve which impairs his efficiency on the project. It may be a day, or a month, but unfortunately, people movement is considered to be transparent to the project completion.

4) Adherence to standards and specifications is either not defined or, if defined, not followed. More often than not, standards do exist, especially in larger installations. They address documentation techniques, labeling, file names, etc. However, once an initial indoctrination is provided for a programmer/analyst, follow-up is ignored. The most expedient solutions are followed, resulting in several steps (modules) in the same program sequence addressing the identical file with different mnemonics. It results for example, in sketchy operations documentation without consideration for restart procedures. Maintenance then becomes a major part of project development.

5) Resource requirements are not anticipated. The major offenders in this area are:

- Data entry. Inadequate time is permitted for turnaround of source code preparation and/or test file operation. Worse, verification may not be performed, which almost invariably adds at least one day to the program development cycle.
- Computer Test Time. The lack of adequate test time becomes extremely critical toward the end of a project, when only one or two programs are being finalized. If turnaround is overnight, each minor change to a program adds at least one full day to the duration.
- Design Level Reviews. Whereas most of the time these are considered in project planning, it is rare that anything longer than a minute is assumed for duration between submission of design specifications and approval.

6) "Brute Force" Approach. In this type of shop, everything is designed and implemented from scratch with no thought given to the use of past projects, tools, or work simplification methods available to shorten the development cycle.

7) Lack of a project manager. It sounds strange, but many projects flounder through to completion without a rudder. The "DP Manager" is normally the project leader and he provides as much attention as he can considering his other duties. In general, very few installations have one man accountable for an entire project, but rather fragment the responsibilities to the point where no one person is accountable.

8) Lack of a Project Log. A project log can be an invaluable tool in performing post-mortems. Further, in companies which charge-back to the user the cost of resources used, it can be the mainstay in justifying such charge-backs.

9) Lack of a project audit trail. Data audit trails are considered the key to the development of any financially

sound accounting system. Yet very few project managers concern themselves with maintenance of a project workbook to provide a similar audit trail for project development.

10) Lack of a skills inventory. Many projects are pursued with the project manager completely unaware of the skills available to him within his own shop. A skills inventory of past accomplishments of each staff member simplifies the staffing of a project and ensures that experience is "recyclable."

11) Lack of project milestones. Because project milestones are not determined at the onset of a project, percentage of completion is usually equated to percentage of hours expended. For example, a project for which 100 hours has been estimated is 60% complete when 60 hours have been expended; when 90% of the hours have been expended, it is 90% complete. This can likewise be extrapolated to 140% complete when 140 hours have been expended.

12) Staff members are considered "universally expert." During the estimation stage, and again during implementation, staff members are considered to be equally competent analysts, designers, programmers, librarians, documentation specialists, etc. They are assigned any of these functions with little consideration given to their ability. Invariably, this results in project delay.

13) Utilization Philosophy. A most fundamental problem which affects many large companies is one which demands maximizing the utilization of personnel, as opposed to a project-oriented approach. When a lull occurs in a particular project, staff members are reassigned, because it is anathema to have people not performing "useful" (that is, design or programming) work. Consequently, when the project restarts, the same people may not be available, or worse yet, are available part time. This is a disastrous approach, because while it assures that people are always assigned to a project and utilization is high, it places an emphasis upon effort, not results.

Termination of the project

In the first place, it is my opinion that projects never terminate. Rather, they become like Moses, condemned to wander till the end of their days without seeing the promised land. However, for those projects that do "terminate," the following are key deficiencies.

1) History/statistics are not determined or not updated. For example, at project termination, the project leader should make some attempt to determine performance in light of certain objectives, or measurable criteria: how many programs were written? how many lines of code generated? average lines of code per day? average source statements per programmer? cpu test time required per programmer? per program? All of the above can be invaluable tools in the estimating and evaluating of future projects. It becomes the first step in the development of a "cost/resource accounting system" for your projects.

2) Quality Control. Typically, when a project is completed, it is never evaluated for quality. The only criteria is "does the program run?" There are no grades (i.e., A, B, C, D, F) of programs. They are either "As" or "Fs."

The manager may evaluate personnel based upon quantity of code, programs or documentation produced, but in fact he never even considers evaluation based upon the quality of coding techniques used.

3) Knowledge gained is rarely transferable. Once a project is completed, it goes through a procedure similar to "de-Stalinization," wherein all vestiges of association with a project are forgotten lest one be stuck with program maintenance. Inadequate time is allowed at the conclusion of the project for staff members to "dump" the knowledge gained or even provide meaningful insight into techniques used.

4) Personnel are not evaluated. There is an ideal time, and only one, to evaluate performance of an individual on a project, and that is immediately at the conclusion of a project. Yet, only too often, personnel evaluation is tied into employment anniversary dates. Between the time an individual has completed a project and his next appraisal, a year may have lapsed. During that year he has had the opportunity to perpetuate mistakes initially made 12 months ago.

5) Lack of formal turnover. Typically, a project termination is first known by the appearance of a new report. More realistically, a formal presentation should take place addressing:

a) initial objectives of the project
b) performance against these objectives
c) review of the end product
d) designation of principal contact for maintenance, etc.

6) Recommendations for enhancement are not documented. At the conclusion of a project (if not earlier) the project team is in an ideal position to recommend enhancements to the system. If these are not quantified immediately, they will be lost forever.

Post termination

The key ingredient here is the conducting of user satisfaction surveys six to nine months after the completion of a project. The survey should address:

a) results versus objective
b) integrity of data
c) freedom from bugs
d) quantification of changes required
e) usefulness of information (i.e., should the system be continued?)

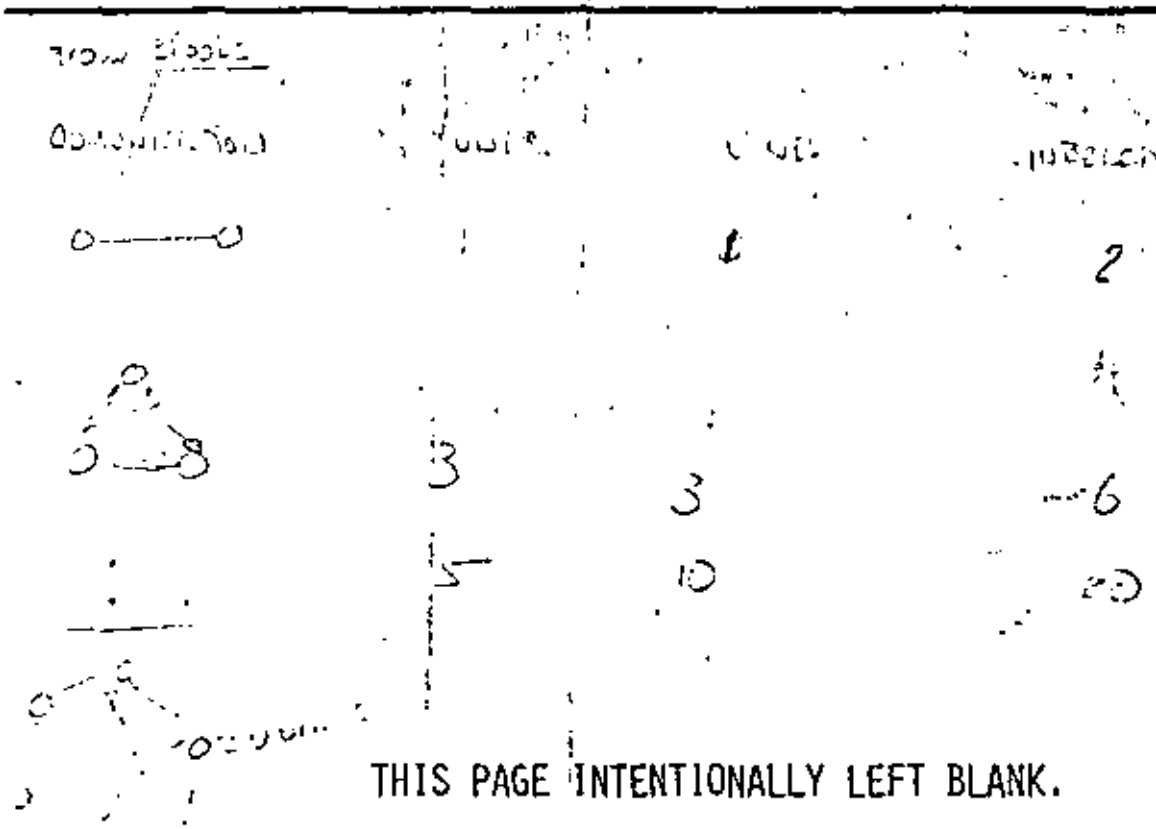
Summary

As a result of reviewing the development of a number of major systems, the above faults exist more often than not. However, the key problems appear in failing to understand the characteristics of a project:

- It has a beginning.
- It has an end.
- It uses multiple, finite resources.
- It has an objective.
- Its success can be measured in terms of time or dollars.
- It requires a leader.
- It requires a staff.
- It must be planned.
- Performance against plan must be reviewed.
- It coexists with other projects but is distinct from them.
- It is measurable (quantifiable).
- It may be a bad project (from the standpoint of usefulness). If it is, it must be altered, or terminated.
- Internal and external forces will affect a project; they must be identified.
- A project is a group of sub-projects.
- No project is unique.

Unless full attention is paid to each of these aspects of a project, the history of project failure will be played out once again.

A vice president and senior consultant with Neoterics, Inc., a Cleveland consulting firm, Mr. Kelder has specialized in operations and systems auditing with emphasis on project management. He held positions in systems engineering, education management, and systems management while with IBM, where he spent 12 years prior to joining Neoterics.



THIS PAGE INTENTIONALLY LEFT BLANK.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

METODOLOGIAS PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ADMINISTRACION DE PROYECTOS DE SOFTWARE

M. en C. Marcial Portilla Robertson

Abril, 1982

PROYECTO "TITANIC"

1. Fecha de entrega.
2. Seleccionar nombre del proyecto.
3. Seleccionar jefe.
4. Desarrollo plan de trabajo.
5. Contratar analistas de sistema.
6. Desarrollar especificaciones generales (sistema).
7. Definir especificaciones de las entredas y orden de la -- programación.
8. Contratar programadores.
9. Comenzar el desarrollo de programas.
10. Comenzar la conversión de programas (archivos maestros).
11. Empezar el diseño de los archivos de datos.
12. Actualizar programas ya elaborados.
13. Contratar mas programadores para cumplir plan de avance ??
14. Comenzar de nuevo a actualizar programas.
15. Comenzar a trabajar
    - Interfaces.
    - Reportes.
    - Documentos.
16. Plan de avance retrasado.
17. PANICO !
18. Contratar mas programadores (el grupo ha crecido 3 veces).
19. Proyecto no terminado a tiempo.

20. Buscar culpable.
21. Castigar inocente.
22. Promover a los no involucrados.
23. Empezar nuevamente (Goto 1).

. " El problema del Software"

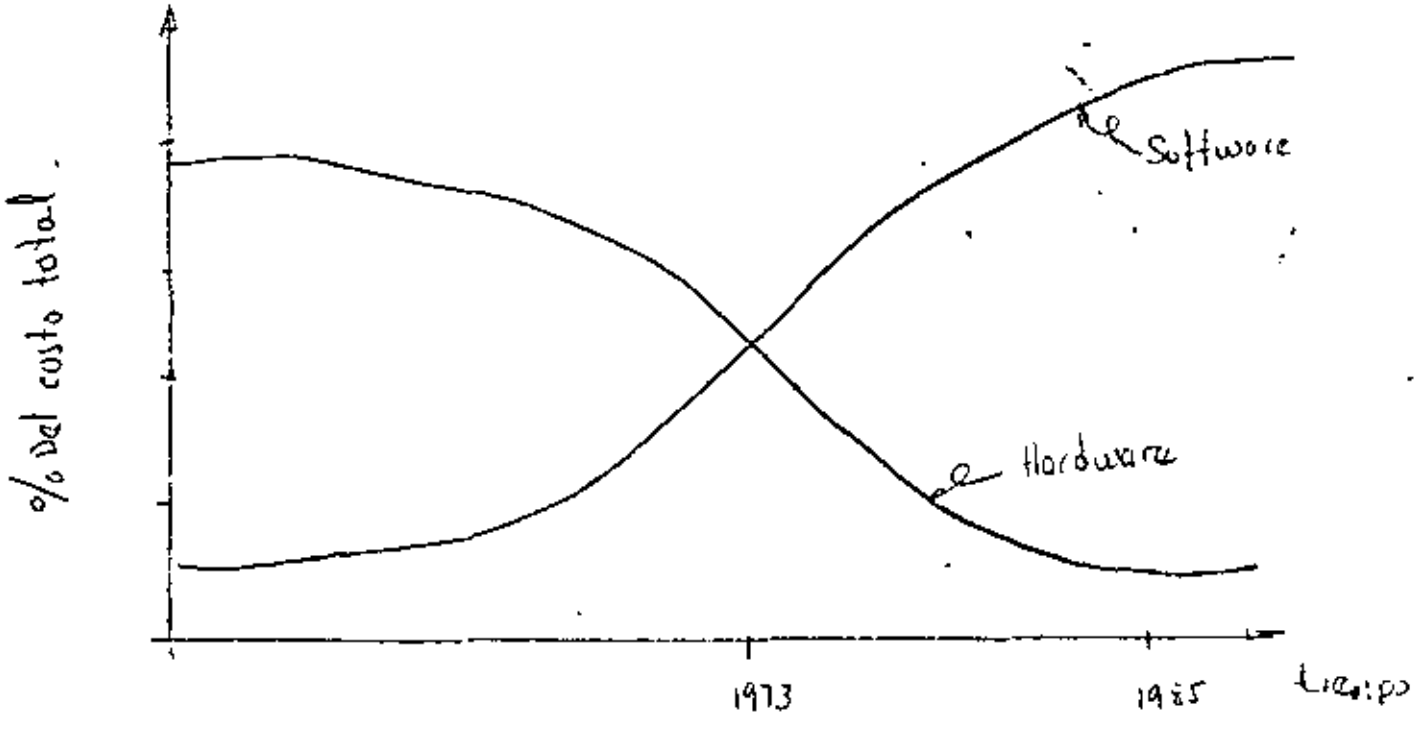
. Estándares
. Nuevas herramientas : Automatización

Todo proyecto de Software tiene: (como todo proyecto).

. Costos de desarrollo.
. Confiabilidad del producto
. Soporte

COSTO DE SOFTWARE.

En los E.U. en 1976 se gastan 80 billones de dólares en escribir software, en 1980 en probable que la cifra llegue a 100 billones de dólares (II-EE procedings sept. 1980).



En 1981 70% Sw
30% Hw

Además la demanda (crecimiento) de software es del 21 al 23% - anual, sin embargo en el mejor de los casos las productividad, y el personal aumentarán el 15% anual.

Para el caso de II (Texas Instruments)

- En 1979 gastaron 100 millones de dólares en desarrollo de -- software.
- Para 1985 habrá 5500 personas en II escribiendo software, -- con gasto mucho mayor al de los 400 millones de dólares.

El software es caro primordialmente por que requiere de trabajo directo (labor) intensivo.

CONFIABILIDAD EN SOFTWARE.

- Para cualquier compañía un"BUG" en un producto terminado es un dolor de cabeza.

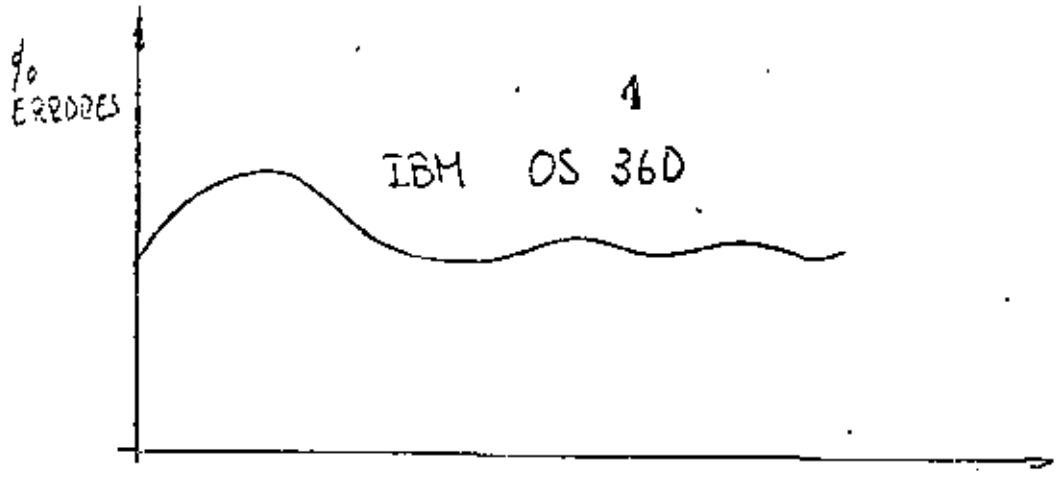
SOPORTE.

- Casi la mitad del 'presupuesto' de 'software' se gasta en - "componer' errores BUGS veamos algunos casos extremos citados- por II.

- En G.M. 75% del presupuesto de sw se gasta en mantenimiento.

60% del presupuesto de GTE (de los sistemas) se gasta en compo ner errores.

76% del costo del IBM 05 360 se gastó en componer errores - - (yourdon).



1 BUG = 0.96 BUG NUEVO

* El peor ——

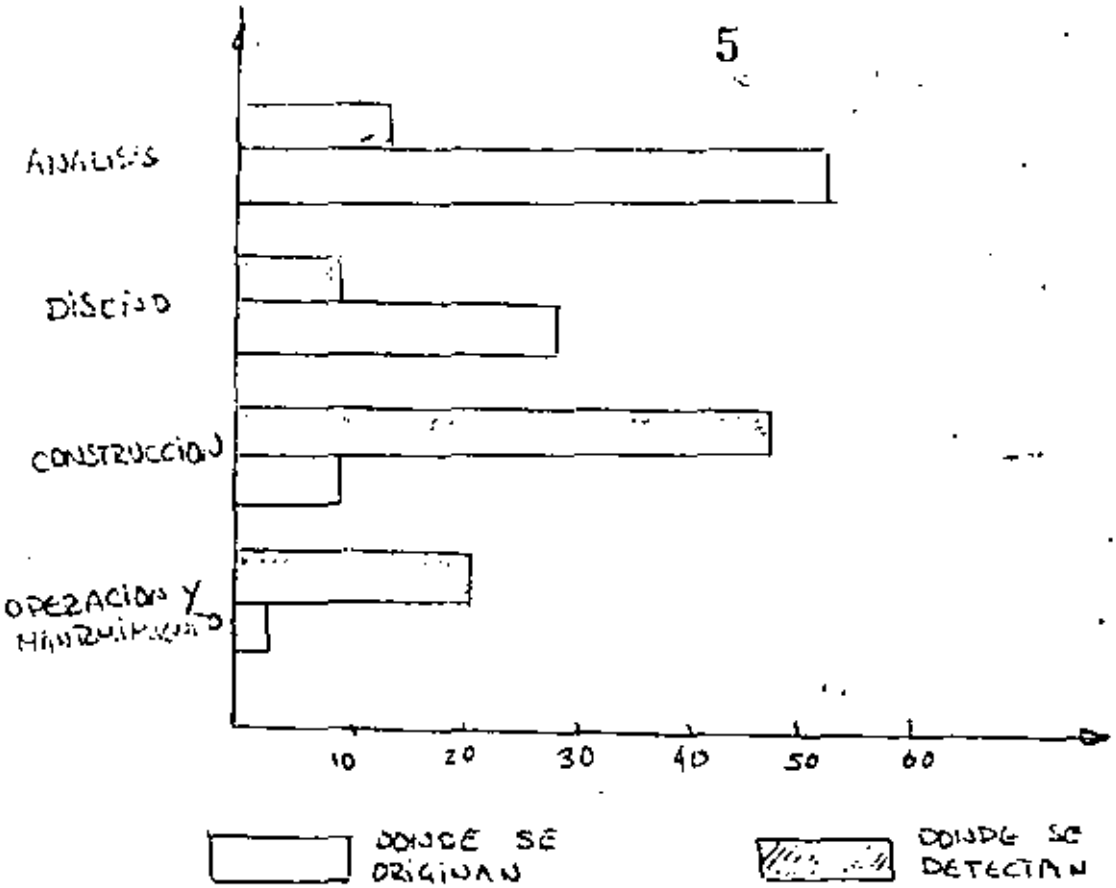
cuesta $4,000 el corregir una línea de código en el 'aeroespacio' y cuenta $75 escribirla.

POR QUE AUMENTAN LOS COSTOS DE SW. (Además de la inflación).

- Resulta difícil diseñar, construir, y probar un programa de computadora, que satisface totalmente al clienta o al usuario.

- Muy frecuentamente el usuario y el analista Subestiman la dificultad del problema.

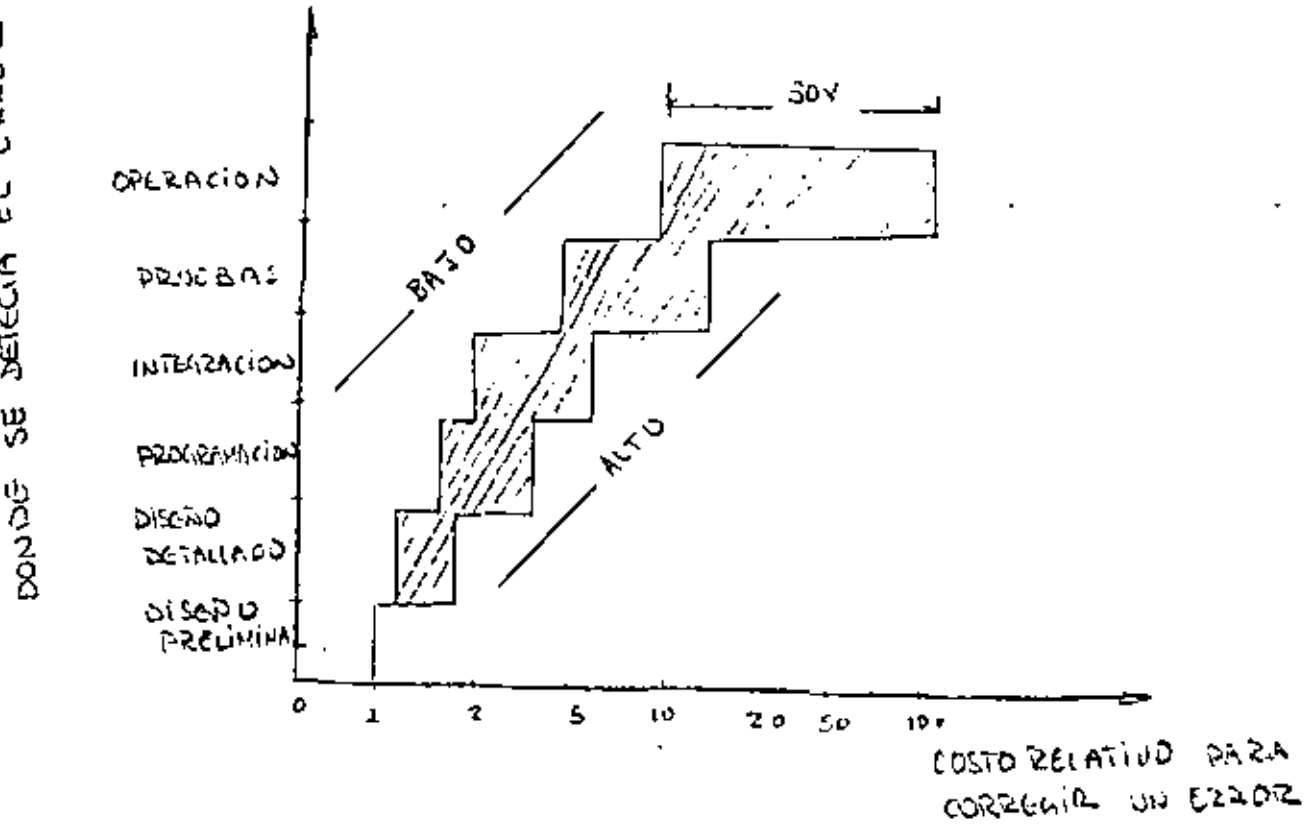
- La mayoría de los errores ocurren antes de empezar a escribir código, y esto ocurre por que se tienen requerimientos muy pobres, y se logran detectar muy pocos de estos errores cuando ocurren.



> 50% De los problemas se originan en la fase de análisis.

Y se corrigen hasta la construcción, lo cual quiere decir cambiar muchas veces el análisis/diseño.

VEAMOS EL COSTO.



Como se ve en la figura anterior aproximadamente 72% de los -- errores se detectan en la construcción o el mantenimiento, y - este error puede costar hasta 30 veces mas corregirlo, que lo- que hubiese costado si se corrige 'donde se origna'.

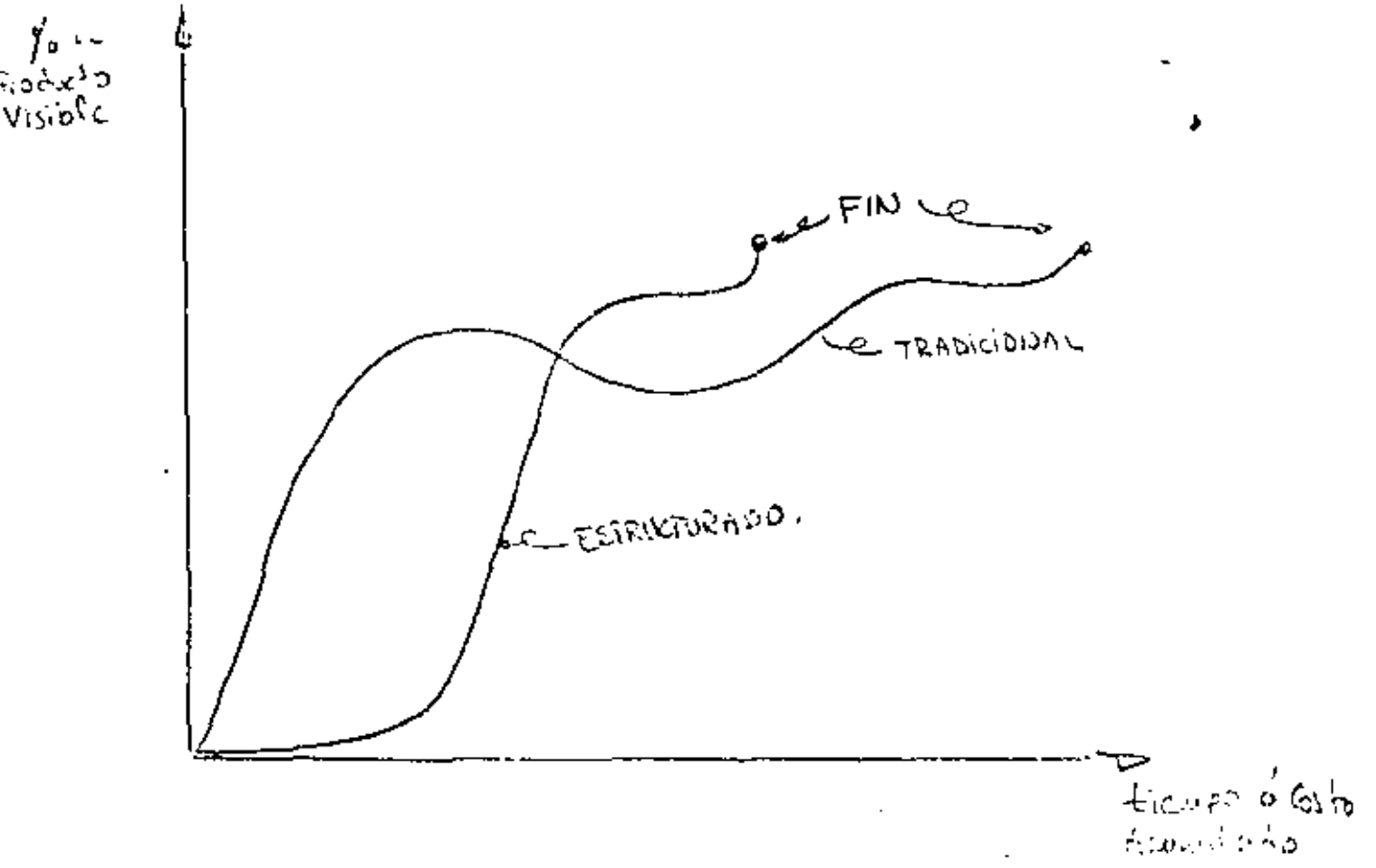
Los errores y las omisiones tienden a multiplicarse con el avan ce del proyecto. Requisitos incompletos, el no entender el -- problema, decisiones vagas, pruebas pobres son las mayores cau santes de los problemas en dos palabras: PLANEACION POBRE.

¿ QUE ES LA INGENIERIA DE SOFTWARE?

La "INGENIERIA" de SW es una colección de métodos y procedimientos para mejorar la calidad del software. Es decir en otras palabras ⇒ poder terminar una tarea de

SW **BIEN** A **TIEMPO** Y EN **PRESUPUESTO**

> Si definimos SW. ENG, podríamos decir que son las reglas para tener buen SW.



Que soluciones ó sugerencias tiene la industria para disminuir los costos de SW y mejorar la calidad.

() MOVILIDAD DE SOFTWARE.

- Compiladores compatibles con los nuevos sistemas.
- Traductores para los Leng. ensambladores.
- Compatibilidad en el desarrollo de herramientas <HW y SW.>
- ESTANDARIZACION.
- Nuevos lenguajes ADA, ACTOR, ACTOR, C, PASCAL.

() REDUCCION DE COSTOS EN MANTENIMIENTO Y DESARROLLO.

- Mejores lenguajes de algo nivel
  - PASCAL
  - C
  - ACTOR
  - ADA
  - FORTRAN 77
- Maximizar la reutilización de software.
- Procurar cambiar el software de hoy por hadware mañana (en PROM, PLA'ER).
- "Desarrollo" de herramientas para el desarrollo y mantenimiento de SW.

() MAYOR CONTROL DE DESARROLLO.

- Programas verificadores de estándares (i.e. CODE CHE -- CKER).
- Programas para seguir actividades (CPM PERT)
- Uso de técnicas de "ingeniería de software"
- Diseño y programación estructurado.
- Mejorar las técnicas de "MANAGEMENT".

() ENTENDER LOS SIGUIENTES CUATRO PUNTOS.

- Recursos limitados.
- Tiempo limitado.
- Creibilidad en el proyecto/solución.
- Avalancha en la complejidad "

"El desarrollar SW para una aplicación específica es uno de los aspectos mas frustrantes del Management".

"Técnicas y rutinas en 'otras ingenierías' resultan inovaciones radicales en SW ingenieri

- T.P. Brooks
"THE MYNTICAL MAN MONDT".

- Antes de continuar hablando de la producción de SW, hay que definir que hace

SW B UENO
B ONITO
B ARATO.

- Debido a que los humanos no se pueden comunicar a la "perfección" se requieren técnicas y metodologías en al 'management' y el control de calidad en el SW a producir.

CARACTERISTICAS DE SOFTWARE DE CALIDAD.

CONBIABLE. - Los programas se comportan tal y como fueron diseñados o c ncebidos?- como se comportan los programas cuando ocurren vi laciones en la entrada?

MANTENIMIENTO. - Se puede leer los programas? existen manuales y documentos en cristiano? se han aislado las funciones de un programa en módulos diferenciables, de manera que un programador pueda fácilmente llevar a cabo el cambio o corregir el error.

PORTABLE. - Se . los programas transportar fácilmente a otros sistemas? son independientes de un HW o con O.S. específicos.

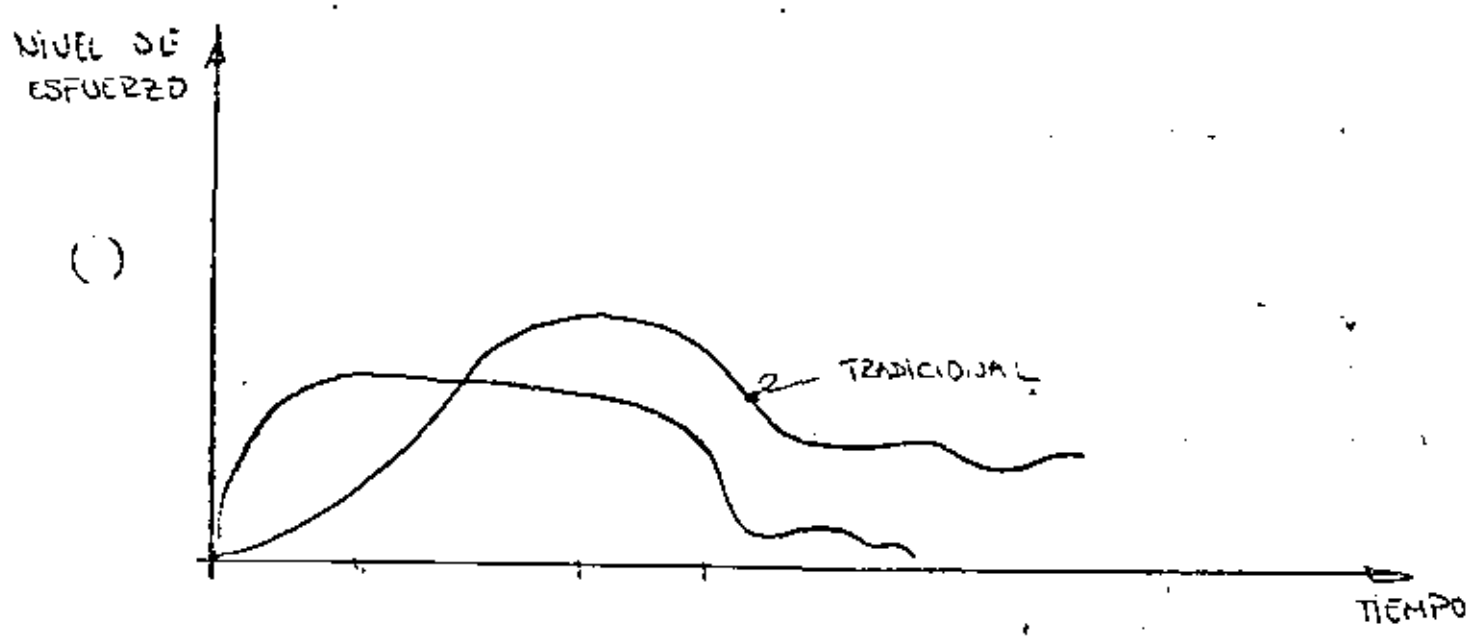
PROBARSE. - Son los programas de estructura simple, de manera que los algoritmos, 1 E/S puedan fácilmente PROBARSE.

EFICIENCIA. - (Economía) - los programas utilizan en formaracional los recursos del sistema.

USABLES - Interfaz hombre máquina.

CICLO DE VIDA DE PRODUCTO DE SW.

El ciclo de vida útil de un programa, debe de ser de varios -- años. Si como se dijo anteriormente el desarrollo inicial es- menor que la mitad de costo total. Y haciendo un poco de historia, sabemos que todo proyecto de SW tendrá cambios y modificaciones continuas es decir tendrá que adaptarse a la dinámica del sistema que vivimos.



Los requerimientos son importantes porque una definición pobre resulta en:

. Fallas
. Omisiones.
. Ambiguedad.
. Inexactivo.
. Sistema nor alizable..

———— Cambios.
———— Altos costos.

VAMOS A DEFINIR LOS REQUERIMIENTOS.

. Los "requerimientos" estan orientados a la estructura del problema.
. Definir tolerancias.
. Tiempos.
. Costos.
. Base de datos.

— Definir las interfaz.

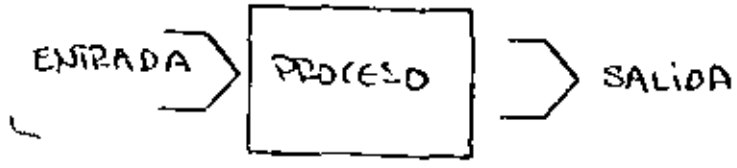
- Usuario.
- Equipo.
- Comunicaciones.

* DEFINIR LOS REQUERIMIENTOS ES DEFICIL.

Hay que dar lugar a cambios futuros
- Evitar acronismos y nemónicos
- Evitar terminología compleja.

° Est diar las necesidades del usuario utilizando metodos "ES -- TRUCTURADOS".

CONCEPTO DE CAJA NEGRA.



. USO DE PROTOTIPOS O SIMULADORES.

. No se esta familiarizando con el problema.
. El usuario no "entiende" el problema.

. HAY QUE REVISAR LOS REQUERIMIENTOS.

. Maximizar eficiencia y confiabilidad a elinimizando costo, memoria, equipo ER. (GUAL).
. No se entiende el objetivo.
. No se cuantifican las metas.

. OBJETIVO DEL PRODUCTO.

( ) CONFIABILIDAD.
  . Tiempo 1/2 de falla.
  . # de fallas.
  . Tolerancias.
( ) PROTECCION DE DATOS E 6S
( ) DETECCION/ ERRORES.
( ) TIEMPO DE RESPUESTA.
( ) USO DE MEMORIA Y PERIFERICOS.

Continuación objetivos del producto.

( ) COMPATIBILIDAD CON.
  . Sistemas anteriores.
  . Lenguajes.
  . Soportes (BSP).

( ) MANTENIMIENTO.
( ) SEGURIDAD Y PRIVACIA
( ) COSTO DEL PRODUCTO.

( ) TRANSFORMAR REQUERIMIENTOS A ESPECIFICACIONES.
( ) PARTICIONAR EL SISTEMA.
( ) PRUEBAS DE LAS ESPECIFICACIONES.
  (-) Entradas
  ( ) Salidas.
  ( ) Intefases.
  ( ) Estructuras.
  ( ) Base de datos.
  ( ) Documentos.

CARACTERISTICAS DE UN BUEN SISTEMA.

  ( ) Jerárquico
  ( ) Modular
  ( ) Rastreable
  ( ) Consistente
  ( ) Implementable
  ( ) Probar

La mente humana sólo "puede" con 7 2
tareas simultáneas

1. ⟹ Particionable.

C A P I T U L O 3

COSTO*, TIEMPO*, CONTROLES*

ESTIMACION.

La estimación es un proceso continuo

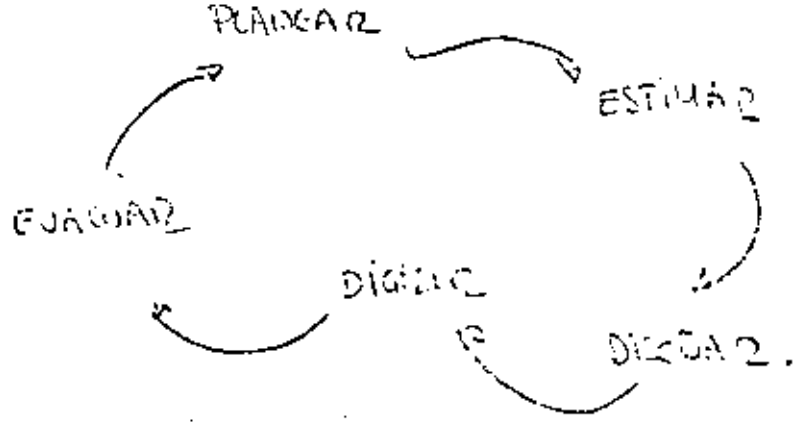
Durante
- La fase de definición
- La fase de diseño
- La fase de integración
- etc.

Esto es cada vez que se cuenta con información

Las estimaciones se hacen mejor con la experiencia.

Como mejorar los estimados.

- Actualizando fechas.
- Llevando una historia de datos (y Libro de Bitácora).
- Llevando control sobre las suposiciones.
- Proyecto estructurado.
- Técnicas estadísticas.
- Software
- Limitando el problema.



Leer Putman "Getting The Software Numbers" IEEE Computer Julio 1980.

METODOLOGIA UTILIZADA EN LA ESTIMACION.

( ) PRINCIPIOS.

- Descomposición del problema en bloques pequeños.
- Estimar el costo y "dificultad de cada bloque".
- Desarrollar un plan del proyecto en detalle.
- Incluir M.O, Equipo, Costos, Admon, Varios, Asesores ER.

( ) VENTAJAS.

- Hay que hacer un análisis detallado.
- Hay que estimar en detalle y con 'exactitud'.
- La estimación se torna en una parte del proyecto.
- Las faltas y errores tendrán repercuciones proventas.

( ) DESVENTAJAS.

- Hay que gasta tiempo y dinero en estimar.
- La actualización requiere de técnicas automáticas.
- ¿ Proyectos pequeños ? Vale la pena.

Si en su plan.

SI FALLA EN PLANEAR, UD. PLANEA FALLAR

PLAN PARA EL CONTROL.

- Costos.
- Calendarios.
- Eficiencia.
- Estándares y C.C.
- Personal.
- Cursos, Asesorias, etc.

ELEMENTOS DE PLAN DE UN PROYECTO.

1. Descripción del proyecto.
2. Estructura de trabajo — De afuera hacia adentro / De arriba hacia abajo
3. Plan de trabajo de cada fase.
4. Entregas (fechas y productos).
5. Organización.
6. Personal — _____ Contrataciones —— Asesores / _____ Cursos
7. Financiamiento (estimar).
8. Equipo.
9. Tiempo y recursos de computadora.
10. Herramientas en SW.
11. Análisis de riesgos.
12. Plan de contingencias.
13. Procedimientos de revisión.
14. Control en los cambios.
15. Plan de integración/pruebas.
16. Programa de entrenamiento.
17. Soportes.
18. Documentos y referencias.
19. Plan de "involucrar al usuario".

PUNTO 2. ESTRUCTURA HIPOS.

ELEMENTOS DEL PLAN DEL PROYECTO.

1. Descripción del proyecto.

+ Breve descripción del mismo.
  - Objetivo, metas.
  - Razón del proyecto.
  - Organización involucrada.
  - Beneficios a obtener.

+ Breve resumen de los requerimientos.
  - M.O.
  - Duración.
  - Costo.

+ Aspectos particulares de interés.
  - Nuevas tecnologías.
  - Situación de competencia insumos.
  - Nuevos métodos de implementar.

+ Entregas.
  - Documentos.
  - SW.
  - HW.

+ Breve análisis de los riesgos.
  - Tiempos.
  - Costos.
  - Insumos.

2. ESTRUCTURA DE TRABAJO - Descripción de cada bloque.

Proyecto	Depto.	Fase	Nomenclatura	Descripción
ALFA	PROG.	DISEÑO	DISEÑO O.S.	Diseñar el OS. del Sist RT. - O.S. Interno - Manual Usuario

3. PLAN DE TRABAJO DE CADA FASE.

* Usar diagramas de GANT o PERT indicando

  - Actividades principales.
  - Dependencias.
  - Eventos importantes.
  - Ruta crítica.
  - Resolución a nivel — Diario / Quincenal / Mensual
  - Entregas.

4. ENTREGAS.

Las "entregas" son el resultado.

* Reportes.

  * Eficiencia.
  * Plan de pruebas.
  * Estudios especiales.
  * Minuta.

* Documentos.

  * Manuales internos — Normas — Codificación / Pal / Pruebas etc.
  * Manuales usuarios — otros documentos ic ruta crítica sv soporte

5. ORGANIZACION DEL PLAN.

——— HIPOS ——— D.F.D. ——— STRUCTURED CHARTS, ETC.

6. ESTIMACION DE PERSONAL. (Ver gráfica)

+ Por individuo.
+ Por actividad.
+ Por paquete

Hacer un cálculo de Hrs hombre y perfil

+ Salarios + asesores
+ Cursos (hasta 30%)
+ Viáticos.

7. ESTIMACION FINANCIERA.

( ) PRESUPUESTO.

+ Año anterior
+ Año presente
+ Año futuro.

( ) DESARROLLAR FORMATOS ESTANDARES PARA FACILITAR ESTA ACTIVIDAD.

+ Desgloce de partidas.

8. EQUIPOS.

+ Rentas.
+ Capital.
+ Desarrollo.
+ Cuando y donde se necesita.
+ Duración de uso.

9. TIEMPO DE COMPUTO Y RECURSOS.

( ) REQUERIMIENTOS.

+ Para gransferirlos al presupuesto.
+ Para utilizarlos en los diagramas de GANT.

( ) UTILIZAR RECURSOS DE LA CAJA ó RENTAR ó COMPRAR.

10. HERRAMIENTAS DE SW.

- Como estimamos tiempo de computadora.
+ Comprar herramienta.
+ Desarrollar herramienta.

11-12 PLAN DE CONTINGENCIAS Y RIESGOS.

+ Que pasa si .....
+ Muchas suposiciones suponen riesgos.
+ Como prevenir riesgos.

13. PROCEDIMIENTOS DE REVISIONES.

⇨ ELEMENTOS CLAVE DE UN PROYECTO.
+ Autitorias
+ Pláticas estructuradas.
+ Rev

+ Que se va a revisar.
+ Quien lo va a revisar
+ Cuando se va a llevar a cabo la revisión.
+ Quien asiste.

⇨ REPORTAR

14. CONTROL DE CAMBIOS.

ELEMENTO CLAVE.

- Por que el cambio.
- Costo de cambio.
- Cambio de calendario.
- Quien aprueba.
- Información necesaria para llevar a cabo el cambio. (Ver procedimientos de la B.S.P.)

15. PLAN DE INTEGRACION Y PRUEBAS.

( ) PREPARAR EL PLAN DE ARRIBA-HACIA-ABAJO.
(Implementar posiblemente mejor que Bottom-UP_
( ) DESCRIPCION DEL PROCESO.
( ) SALIDAS (QUE DEBEMOS VER). Alarmas
( ) INTERFAZ HOMBRE-MAQUINA Acciones
etc.
( ) RESPONSABILIDADES

16. PROGRAMAS DE ENTRENAMIENTO

Seminarios, cursos (lugar, costo, tiempo)
Conferencias.

( ) DEFINIR UN PROGRAMA DE ENTRENAMIENTO.

17. PLAN DE SOPORTE.

- Una vez terminado un producto hay que soportarlo.

  - Mantenimiento
  - Cursos.
  - Garantias, etc.

18. DOCUMENTOS Y REFERENCIAS.

19. PLAN DE INVOLUCRAR AL USUARIO.

DISEÑO.

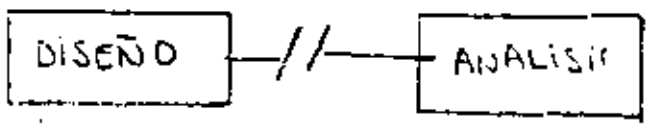
1. Aspectos directivos.
2. Elementos de diseño.
3. Consideraciones especiales.

1. Aspectos Directivos.

- Diseño.
- Organización.
- Personal.
- Documentos.
- Revisiones sobre el diseño.
- Control en los cambios.

1.1 Diseño.

El diseño se inicia una vez que fue aprobado 'lo que no se va a producri' y el plan de implementación. Se especifica como se 'rompe' cada función en partes, estas partes serán los módulos a programar.



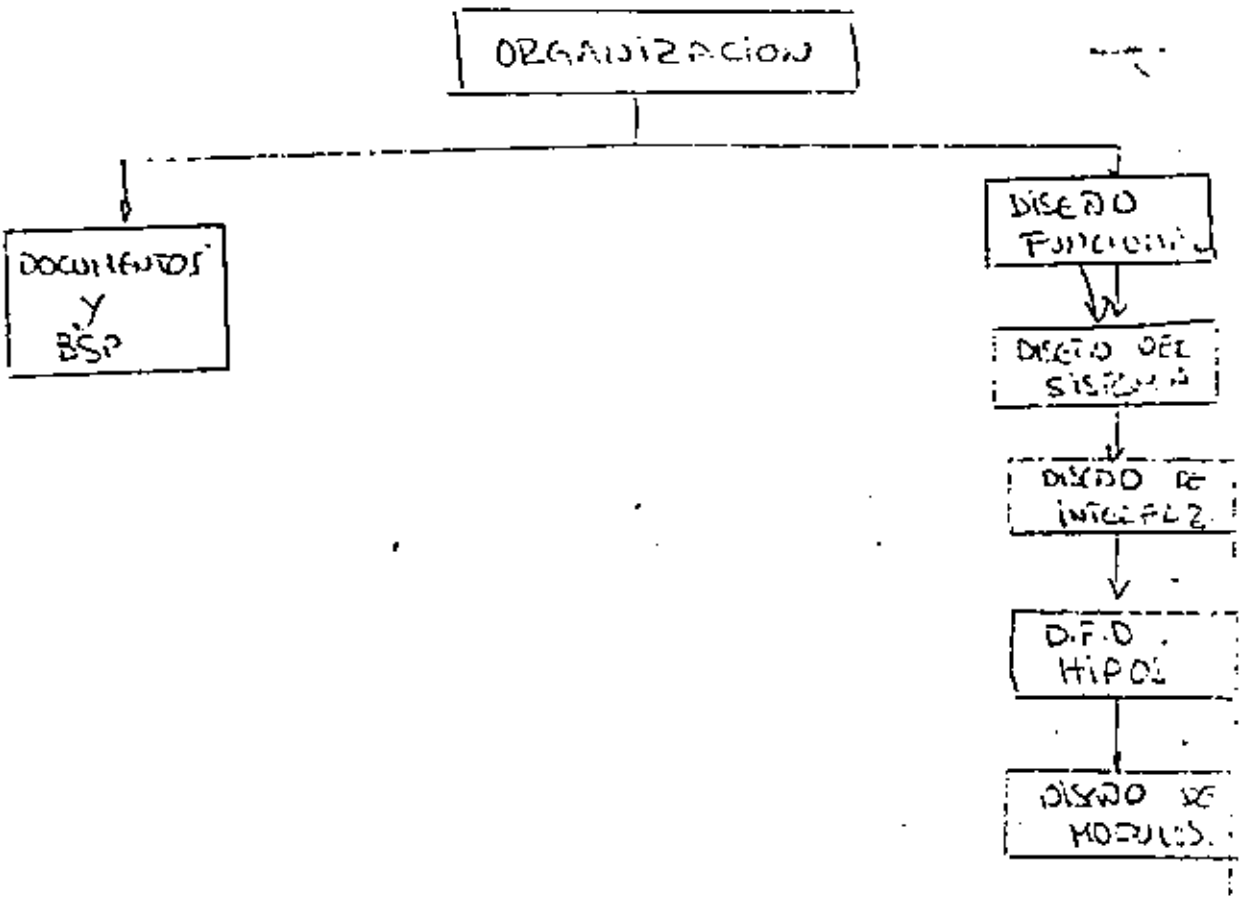
- Deben 'direccionar' todos los aspectos del sistema (sub-sistema)

Algunas preguntas que nos tenemos que responder son:

- Que tan grandes son los módulos.
- Como dividir el diseño y la implementación.
- Como y que vamos a probar.
- Límites en: Memoria
  Disco
  Tiempo CPU
- Hay que modelar ó simular.

Al final del diseño debemos saber que vamos a producir, las especificaciones, así como un plan de pruebas.

1.2 ORGANIZACION.



DISEÑO.

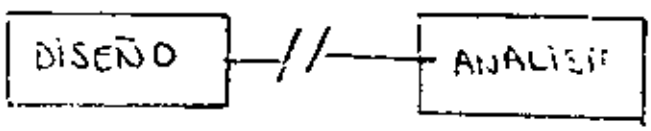
1. Aspectos directivos.
2. Elementos de diseño.
3. Consideraciones especiales.

1. Aspectos Directivos.

- Diseño.
- Organización.
- Personal.
- Documentos.
- Revisiones sobre el diseño.
- Control en los cambios.

1.1 Diseño.

El diseño se inicia una vez que fue aprobado 'lo que no se va a producri' y el plan de implementación. Se específica como se 'rompe' cada función en partes, estas partes serán los módulos a programar.



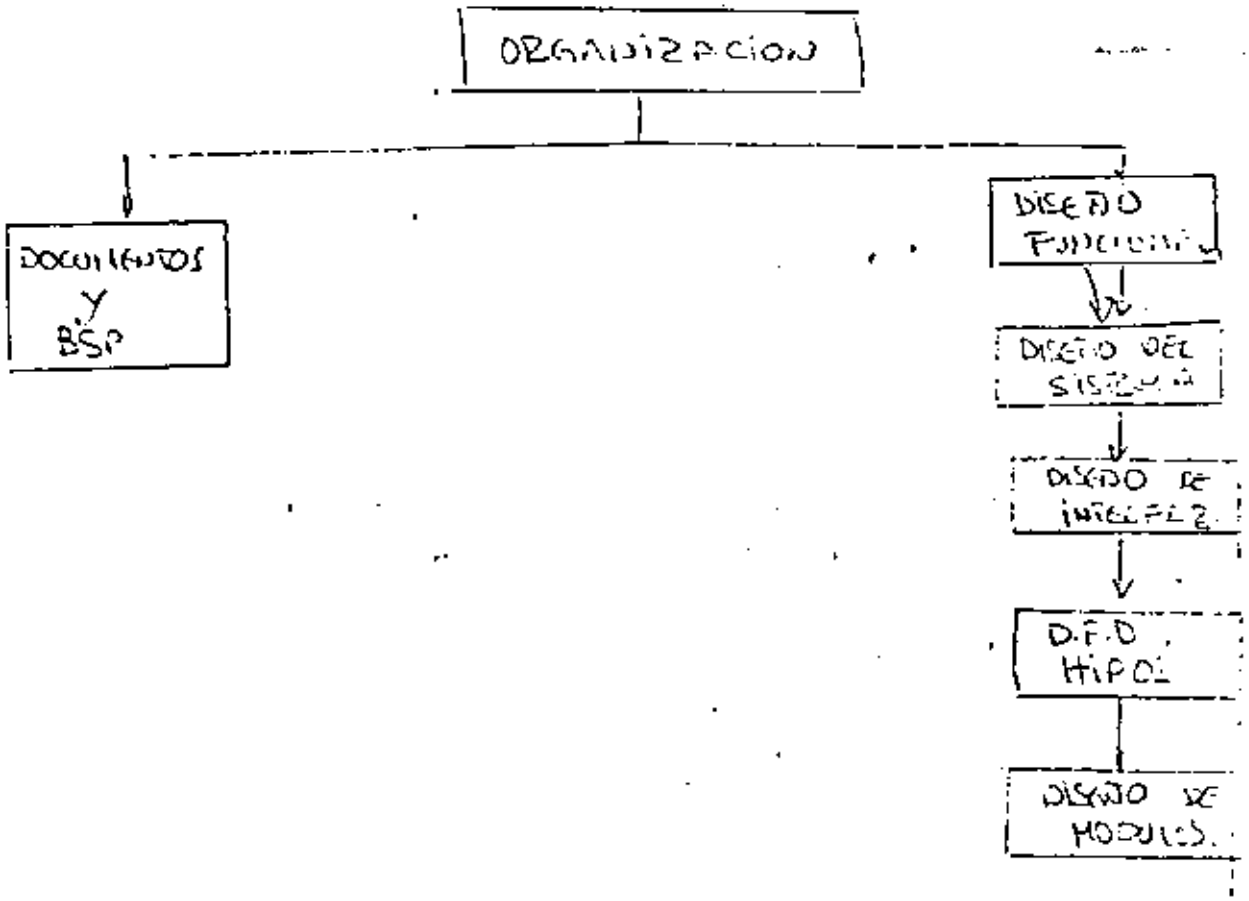
- Deben 'direccionar' todos los aspectos del sistema (sub-- sistema)

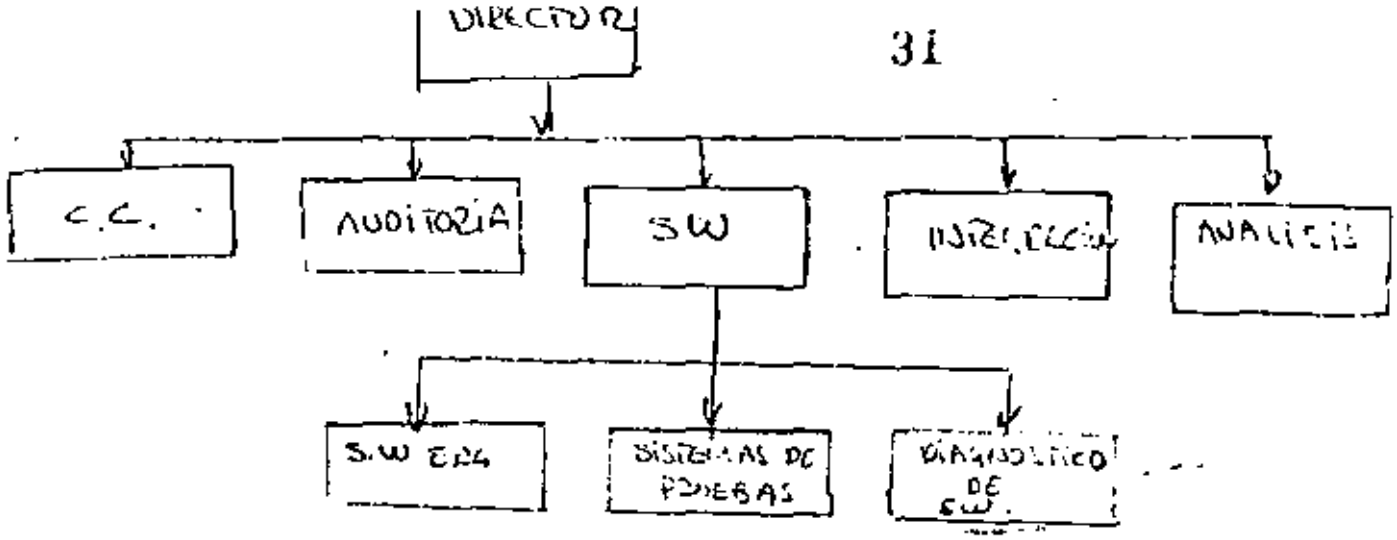
Algunas preguntas que nos tenemos que responder son:

- Que tan grandes son los módulos.
- Como dividir el diseño y la implementación.
- Como y que vamos a probar.
- Límites en: Memoria
  Disco
  Tiempo CPU
- Hay que modelar ó simular.

Al final del diseño debemos saber que vamos a producir, las especificaciones, así como un plan de pruebas.

1.2 ORGANIZACION.





PERSONAL.

+ Diseño funcional
  - . Orientado al producto.
  - . Orientado al sistema.
    - Programadores
    - Consultores.

+ Diseño de sistemas
  - . Con experiencia arcos.
  - . Con experiencia en programación

  Problemas Modelo
  Síntesis
  Análisis

. DOCUMENTOS.
. REVISIONES.

- . Calendario de revisiones
- . En la revisión se deben tener los documentos (sistemas) a revisar antes de la sesión.
- . No transforme las sesiones de revisión en sesiones de diseño.
- . Anote los problemas.
- . Asigne fechas y responsables.

ELEMENTOS DE DISEÑO. (EN GENERAL)

( ) LEER EL DOCUMENTO DE LOS REQUERIMIENTOS.
( ) EXTRACTAR LOS REQUERIMIENTOS 'CON INFLUENCIA EN EL SISTEMA' Escribir uan especificación funcional
( ) DISEÑAR LA ARQUITECTURA DEL SISTEMA.

——— Especificaciones del sistema.
——— Interfases, flujo de datos, e/s, estructura
( ) Diseñar los módulos (y sub-sistema
. Especificación del módulo (y subsistema.

AL FINAL DEBEMOS TENER UN DOCUMENTO
" LA ESTRUCTURA INTERNA DEL SISTEMA-SUBSISTEMA"

CARACTERISTICAS DE UN "BUEN" DISEÑO DE SISTEMAS.

Modular.
Responsiva.- El sistema debe responder al llamadas exteriores
Eficiente
Mantenible.
Modificable.
Comprensible. (No se debe requerir un Dr. para entenderlo).
Probable y
Transportable.

UTILIZAR EN LA DESCRIPCION DEL PROBLEMA.

1. DIAGRAMAS DE ESTRUCTURA.
2. HIPO'S.
3. DIAGRAMAS DE FLUJO DE INFO.
   (Burbujas - Yourdon)
4. LENGUAJE PARA DESCRIBIR PROGRAMAS
   PDL o PSEUDO (o Dino)

NO UTILIZAR. DIAGRAMAS DE FLUJOS

MEJOR ó EN LUGAR DE:

WARNIER — Los veremos un poco mas adelante*

ó

NASSI — SCHNEIDERMAN

* Tomado del curso 'Desarrollo de Sistemas' F.A. Rosenblueth.

- Y utilizar métodos modernos de análisis, diseño implementación o prueba.

- Permite actividades en paralelo.

- Permite retroalimentación y la oportunidad de repetir 'algunas' actividades.

EJEMPLO DE PDL (CASI PDL)

SUBRUTINA SCIIA EBC

LEE Longitud del Buffer de entrada en Bufent

$ PTR+0

LLAMA el MGR del espacio libre para que de un Buffer igual a Longitud (BUF. Salida)

IF

SI no hay Buffer ENTONCES regresa con + no BUFF.

ELSE HAS HASTA todos los caracteres del buffer de entrada se han convertido

READ

LEE los caracteres ASCIL del buffer de entrada un INBUFF $PRT + 1

Convierte a EBCIDIC (luego decidimos el algoritmo)

WRITE

ESCRIBE los caracteres EBCIDIC and buffer de salida.

FINDO

CALL

LLAMA al MGR del espacio libre para que 'deje' el buffer de entrada

Regresa el estado = termine y OUTBUFF$PTR

Utilizar solamente las siguientes estructuras

CONSIDERACIONES ESPECIALES

° El SW debe escribirse para el FUTURO

° La transportabilidad requiere consideraciones especiales

° Hay que diseñar con la idea de MANTENER

Veamos esto

Utilizar solamente las siguientes estructuras

Costos de Desarrollo

FASE DE IMPLEMENTACION

° ORGANIZACION

° ELEMENTOS DE LA IMPLEMENTACION

En la implementación de 'grandes proyectos. se recomienda

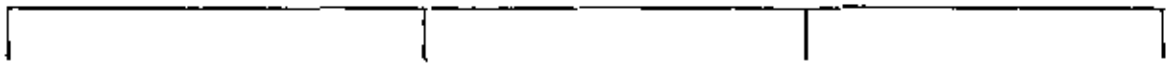
° Organizacion Funcional por Subsistema y tipo

( ) Lo que mantiene modulos logicos agrupados

( ) Minimiza el 'SPAN' de las comunicaciones

( ) Mejora el 'inventario'

i.e.



SISTEMA	DIAGNOSTICO	HERRAMIENTAS	COMUNICACIONI (SW)
- OS	+ COMPONENTES	+ TRADUCTORES	+ SDLC
- DRIVERS	+ PERIFERICOS	+ INTERPRETES	+ BISYNC
- UTILERIAS DEL SISTEMA		+ EMULADORES	+ ASYNC
- CONTROL ARCHIVOS		+ REVISORES DE CODIGO	+ AUTO ANSWER PISTL
DATA BASE		+ SIMULADORES	

GENTE (El mayor problema)

Fijese usted: No hay un tipo de programadores

+ Administradores	+ matematicos	+ Musicos
+ Informaticos	+ Actuarios	+ Artistas
+ Ingenieros	+ Psicologos	

POR LO QUE:

+ Existen problemas de comunicacion entre los ≠ grupos y personas con diferentes antecedentes.
+ Se hace difícil la selección de personal para los puestos de mando
+ Se tienen diferentes habitos de trabajo

+ Personas que conocen

+ Aspectos de HW
+ Aspectos de SW
+ .Aplicaciones
+ No saben ensamblar
+ ETC

POR LO QUE:

+ Hay que capacitar
+ Hay que fijar STD'S y NORMAS

CAPACITAR
+ Se involucran las personas con la empresa
+ se mejoran las aptitudes
+ se mantiene el personal capacitado

STD'S
+ Documentos
+ Programas
+ Procedimientos
+ Reportes
+ Planes
+ Normas
+ Diseño
+ ETC

META:

Construir un sistema que funcione

- Por pasos/Etapas
- En cada etapa "hay que demostrar"

FILOSOFIA PARA IMPLEMENTAR

TAMAÑO DEL PROYECTO

+ Los proyectos pequeños se pueden implementar con mayor facilidad hay que romper en modulos
+ Mejor control del personal en proyectos pequeños

SI PODEMOS ESCOGER

+ Tener el HW Corriendo
+ Tener el S.O. Funcionando
+ Tener las utilerias y herramienta de desarrollo

HAY QUE

Minimizar el tiempo/costo de desarrollo

Minimizar utilizacion de HW

Maximizar el mantenimiento

CODIFICACION ESTRUCTURADA

LENGUAJES

TECNICOS	Algol	
	Fortran	OLD
	Basic	
ESTRUCTURADOS	Fortran ?? (?)	y PL/1(?)
	Algol	
	Pascal	
	ADA	
Orientados a cada maquina		Ensamblador

Que lenguaje utilizar

- Minimo tiempo de desarrollo — PL/1, PASCAL, ADA
- Minima utilizacion de Memoria

  Rápida ejecución — Ensamblador

CODIFICACION ESTRUCTURADA:

- Genera modulos que

1.- Son relativamente pequeños

° alrededor de 60 a 100 líneas de codigo incluyendo comentarios

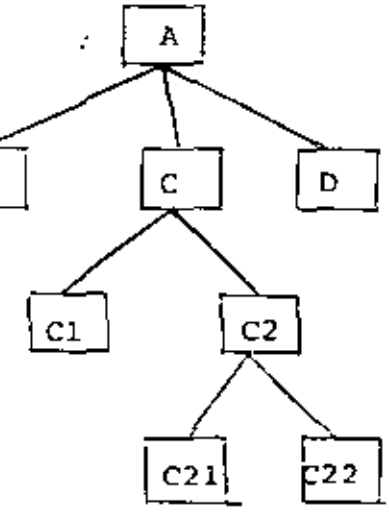
° Se puede 'pensar' en todo el algoritmo

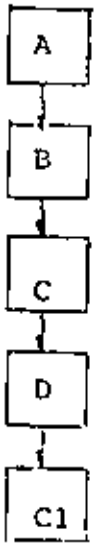
2.- Relajadamente acoplados

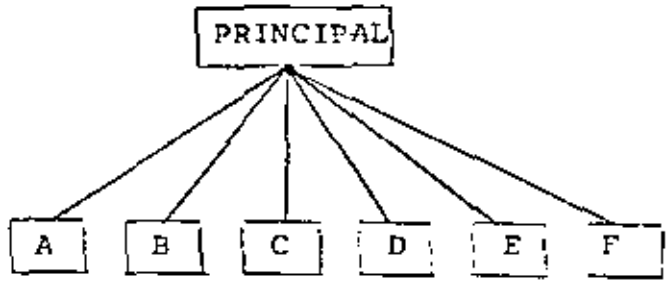
° Una sola entrada, llamadas simples a rutinas

° una sola salida, pasar parametros en forma simple

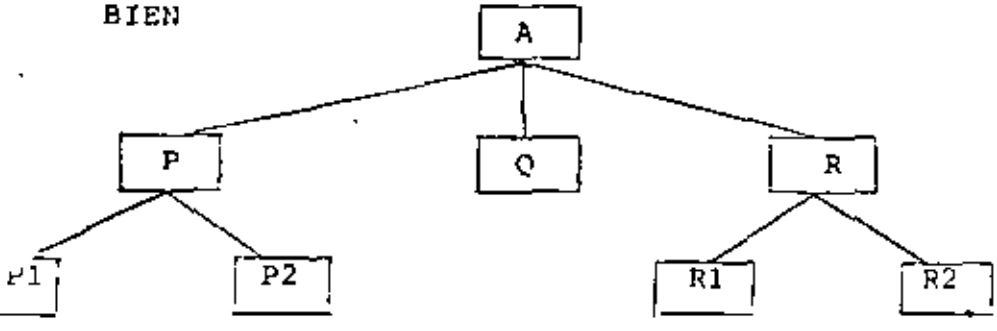
i.e. MAL







BIEN



° Acoplamiento es una medida de la "fuerza" de asociación entre modulos

° NO queremos modulos fuertemente acoplados

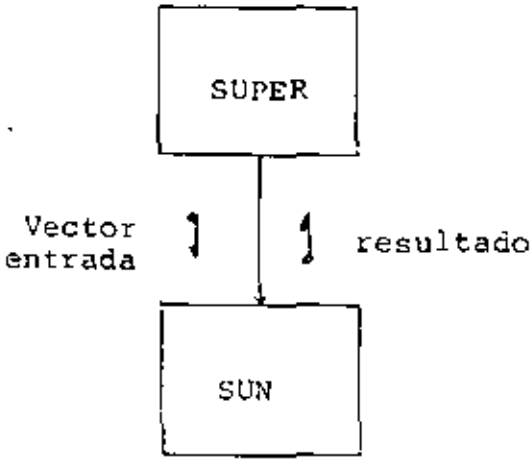
° Si modificamos un modulo queremos NO tener que modificar otros, es decir minimizar el 'impacto'de las modificaciones

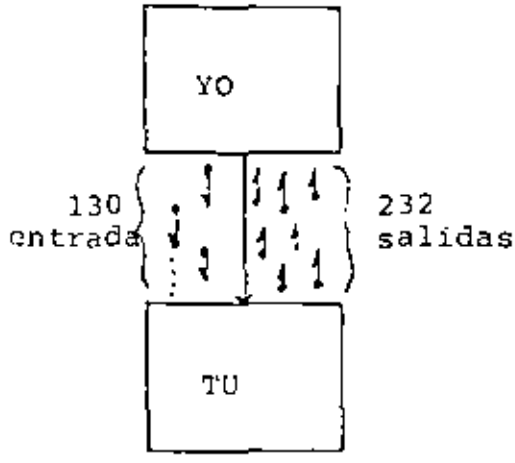
Es decir debemos poder mantener o corregir un modulo sin tener que "saber nada" de lo que pasa en los otros.

Los Factores que tienen influencia sobre los modulos son (Acoplamiento)

- La conección entre los modulos
- La complejidad en la interfaz
- El tipo de flujo de información

Las correcciones normalmente significan un "bajo" acoplamiento versos conecciones patologicas





Problemas con COMMON

. Common en Fortran
. variables globales PL/1 ALGOL,C
. Data Division en COBOL

4.- Funcionalidad

°Los modulos deben ejecutar funciones simples y relacionadas

°Se deben poder definir en terminos de una caja negra (características externas)

5.- Codigo documentado

ENCABEZADO

Nombre del Programa

Fecha Revisión

Programo Reviso

Fecha

° Objetivo

° Entradas

° Salidas

° Logicas

° Externas

°

PDL

6.- Organización Consistente

- Encabezados en la documentación
- Externas
- Publicas
- Declaraciones de datos
- Salidas
- Subrutinas

Fuentes de error (Ruben, Dana y Biche ieee, Trans on SW eng. Junio 25)

Error en la especificación	28%
Desviación intencional de la especificación	12%
Violación de los std's de programación	10%
Error en los accesos a datos	10%
Error en la seguenciación ó decisión logica	12%
Error en calculo aritmético	9%
Tiempo invalido (invalid timing)	4%
Manejo de interrupciones	4%
Constantes o valores de datos erroneos	3%
Documentación inexacta	8%

¿ QUE NOS DICEN ESTOS DATOS?

- errores en la fase de desarrollo

ALGUNA FORMA DE EVITAR LOS ERRORES EN LA FASE DE DESARROLLO?

- campaña de platica
- involucrar al usuario

EL USO DE
HERRAMIENTAS

"ENGINE"
- Compiladores
- Ensambladores
- Linkers
- Utilerias

LIBRERIA
- Libreria codigo fuente, programas
- Libreria codigo objeto
- Documentos
- Materiales de Soporte

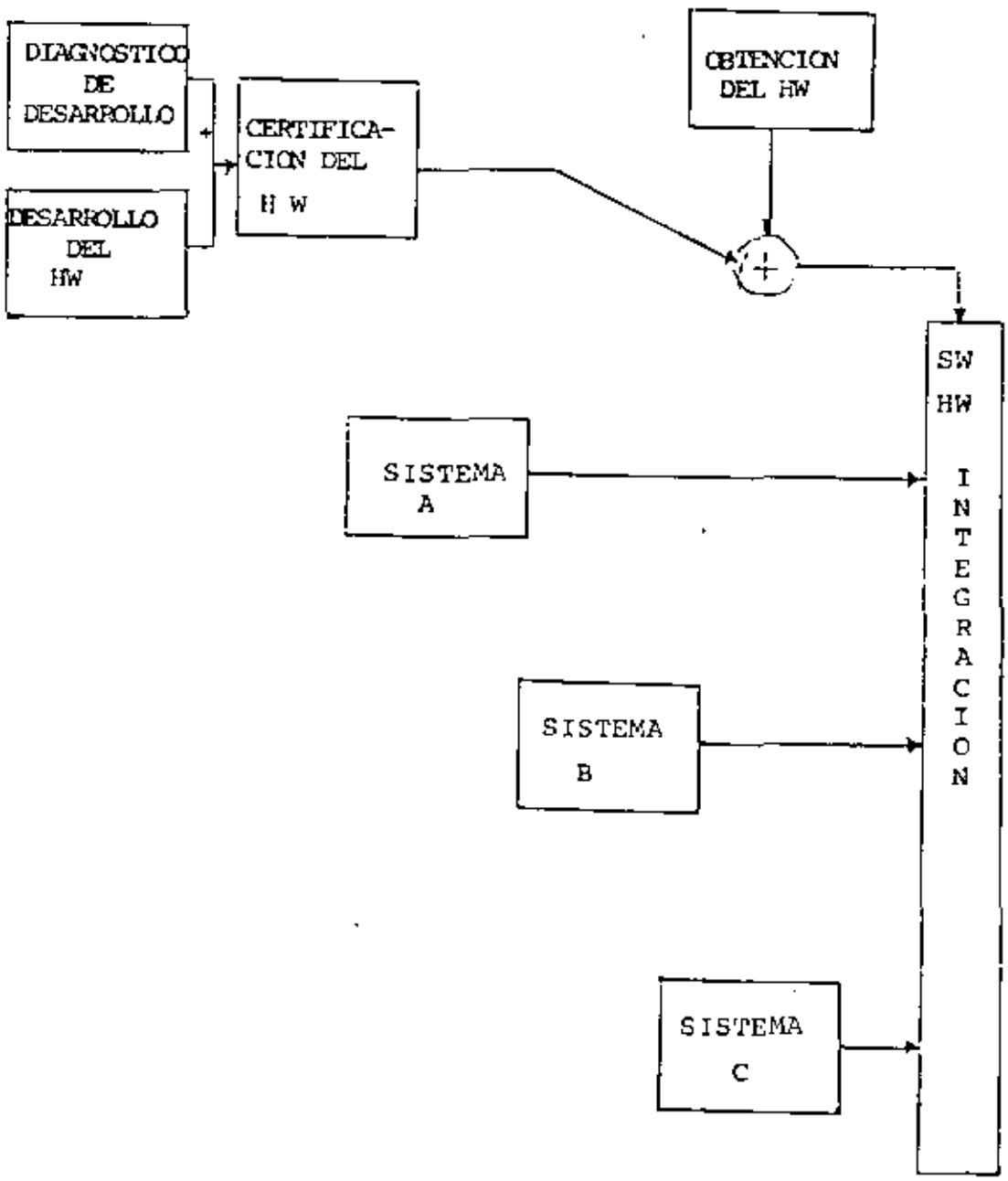
Emuladores (ICE): = (IN-CIRCUIT-EMULATOR)

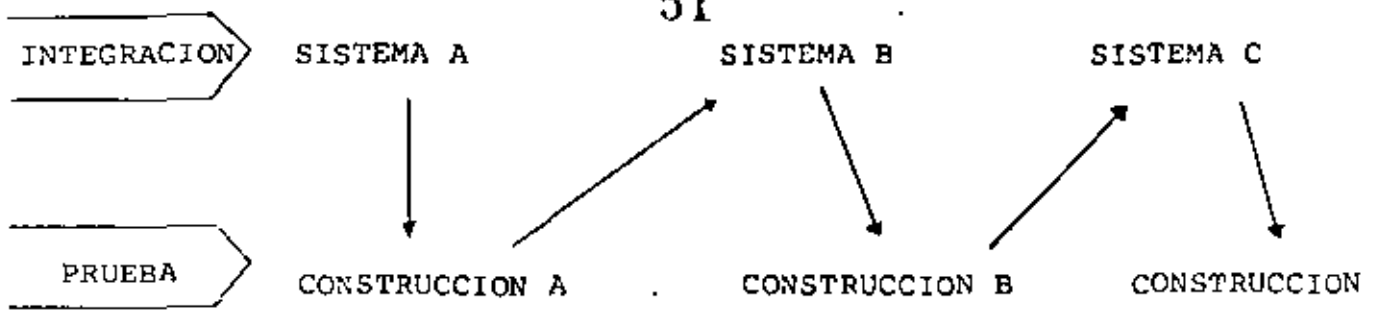
- De BUGGER para tiempo real (Software)
- De BUGGER para tiempo real
- Programadores PROMS

FILTROS

FASE DE INTEGRACION

"Desde el punto de vista de SW la manera ideal de integrar SW es hacerlo en HW totalmente operando (Versión a utilizar)





La integración involucra 3 aspectos principales:

( ) El proceso de integración (de acuerdo al plan)

- referenciamiento de la unidad probada a la libreria
- hacer link' de los modulos para tener una nueva construcción.
- preparar la transmisión documentación de liberación

( ) El proceso de prueba

+ 'correr' la construcción de acuerdo a la 'prueba'
+ hacer diagnostico y "atrapar" errores
+ Generar un reporte de la construcción

( ) Proceso de evaluación

+ Evaluar los resultados de la prueba
+ Decidir si se acepta, para llevar a cabo la siguiente construcción

Que pasa si hay errores de diseño o

la integración

ie utilizando PDL

PROCEDURE: PRUEBA

TOMA Construcción de la Biblioteca

Verifica la construcción - n si cumple con los stds.

SI no-standar 6

entonces rechaza la construcción y espera una nueva construcción

Else inicia los procedimientos de prueba / congela la construcción

TERMINA SI

DO mientras 'los errores no son muy grandes y las pruebas no se han terminado CORRE la construcción -n utilizando el bloque de prueba- n

DEBUG y prepara retroalimentando al grupo de implementación

TERMINA NO

Genera reporte evaluador

Espera la siguiente construcción

TERMINA REST

+ Recuerde que por razones de 'magia' los errores salen en la demostración al cliente y estos son peores si se trabaja en linea

Por lo que

( ) En las pruebas de diseño, para cada elemento se debe utilizar una matriz de prueba correspondiente de prueba indicando la función.

+ La operación de cada función es calificada de "correcta" si cumple la prueba cuando se corre.
+ Los problemas se deben reportar en la forma adecuada.

( ) La matriz de correspondencia debe incluir

+ Operaciones así como rangos
+ operacion con entradas anormales o cursos extraños
+ operación por actividad

LA EFICIENCIA SE MIDE

i.e. El sistema ordenará nnn registros del tipo yyy en zz seg.

El sistema mantendrá una exactitud de X%
una temp de y C
la respuesta de la servo valor w es de y seg
etc.

Si no podemos medir ó especificar NO se incluye el parrafo en las especificaciones.

( ) El diagnostico de SW es similar al SW de aplicación
+ tengo un requerimiento que satisfacer
+ existen especificaciones
+ se planea su desarrollo

( ) El diagnostico de SW es diferente al SW de aplicación
+ es un esfuerzo en conjunto de HW/SW
+ La integración inicial se puede llevar a cabo en HW diferente o HW sin haberse probado (nuevo)
+ la verificación requiere conocimientos de HW.

Se deben tener 3 modos (cuando menos)

MODO 1	• FALLA	El sistema va a tener problemas
MODO 2	PREDICCION	El sistema puede tener problemas
MODO 3	OK	El sistema deberá correr OK

Cuando falla sucede lo que se conoce como

LOCURA DE CAMBIO

Hay que justificar todos los cambios

- no existe el cambio pequeño
- cada cambio da lugar a una nueva construcción
- los errores deben eliminarse hay que hacer cambio

LOS CAMBIOS SE HACEN A TRAVES DE LA BIBLIOTECA, DE SOPORTE DE PROGRAMAS

LEY DE BLOOK'S

Si su proyecto esta atrasado y contrata mas personal para salir adelante, mas tiempo se atrasara y esto se cumple como que 1+1 = 0 y llevamos 1

METODOS DE PRUEBA

- °Pruebas convencionales
- °Plan de pruebas
- °Generación de datos de prueba
- °Pruebas de aceptación

No vamos a discutir "pruebas de corrección de programas" (todavía no se descubre o se inventa bien)

PRUEBAS CONVENCIONALES

- ° Combinan 50% del tiempo y recursos de algunos proyectos
- ° Se hacen de abajo hacia arriba
- ° Terminan con la paciencia de muchas personas (especialmente con la del jefe)
- ° Gran cantidad de prueba y baja calidad.
- ° Poco control de la dirección.
- ° 1 a 5 errores/100 líneas de código.

DESARROLLO DE UN PLAN DE PRUEBAS

Punto básico.- Sin un plan organizado no se pueden tener pruebas 'decentes' por lo que

1.- Responsabilizar a una persona/grupo de las pruebas

2.- Establecer los procedimientos y los STD'S para llevar a cabo las pruebas

3.- Describir los casos de prueba.

- para aceptación
- y versión de arriba hacia abajo

4.- Criterios de terminación de pruebas

5.- Calendario de actividades y recursos requeridas, para llevar a cabo las pruebas

MANUAL DE ESTANDARES VOL 31 COMO CONSTRUIR CASOS DE PRUEBA	MANUAL DE ESTANDARES VOL 32 COMO DOCUMENTAR Y USO DE XXX	MANUAL DE ESTANDARES VOL. 32 CONVENCION DE NOMBRES	MANUAL DE ESTANDARES VOL 3 BASE DE DATOS

PROCEDIMIENTOS Y STDS PARA PRUEBAS

Para poder llevar a cabo las pruebas de una forma 'manejable' uniforme se debe

- Tener un esquema para la de los casos de prueba
- Documentación de los casos de prueba y resultados
- Convención en el uso de los nombres de las variables
- STD'S para el uso de la base de datos

Responsabilizar a una persona o equipo de las pruebas

- Dependiendo del tamaño del proyecto se debe tener un # de personas en las pruebas.
- Por razones obvias, deben de ser personas ≠ a las que desarrollaron

Probar es el proceso de ejecutar un modulo con el fin expreso de encontrar errores.

GENERACION DE DATOS DE PRUEBA

- Para generar datos de prueba 'inteligentemente' debemos considerar:
  - que aspectos del sistema (comportamiento) son los que vamos a probar
    - funciones externas
    - eficiencia
    - logica interna
    - etc.

Esto nos determinará los valores de los datos de prueba

- Que "combinación" de los componentes del sistema estamos probando.
  - modulos
  - colecciones de modulos
  - modulos con E/S
  - modulos con acceso a B.D.

Esto nos determinará la complejidad de las pruebas

+ ¿Como podemos utilizar herramientas automatizadas para mejorar la eficiencia y productividad del proceso de prueba?

Hay que limitar las pruebas

- linites de E.
- limites de S.
- limites funcionales
- loops
- condicioens invalidas.

POR EJEMPLO, SI AL PROBAR UN PROGRAMA "ORDENARTABLA" QUE PASA SI

- La tabla esta vacia
- solo se tiene una entrada
- todos los elementos son iguales
  la tabla esta llena
- los elementos ya estan ordenados
  etc.

PRUEBAS DE ACEPTACION

EL USUARIO

- Hay que involucrar al usuario
- las pruebas deben salir solamente de las especificaciones (RPC)
- Si al sistema no pasa las pruebas de aceptación, seguramente hay una falla en el Análisis y no en la Codificación

RESUMEN DE SW ENGINEERING

UN PROYECTO DE SW TIENE 8 FASES QUE SON:

I INICIO

II DEFINICION

III DISEÑO

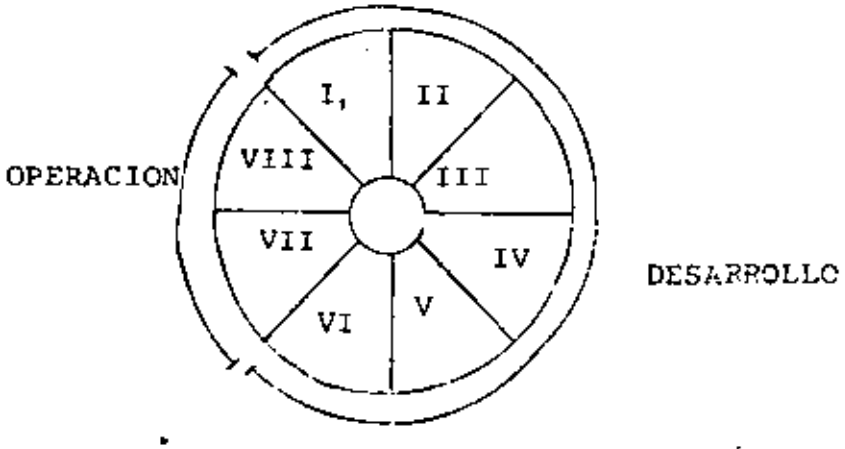
IV PROGRAMACION

V PRUEBA SISTEMA

VI ACEPTACION

VII INSTALACION

VIII OPERACION



FASE DE INICIO.- En esta fase hay que contestar:

- Cuales son los requerimientos
- Que se debe producir
- Quien es el usuario, cual es el mercado
- Cuales son las ventajas/desventajas
- Que características y efectos va a tener
- Es factible
- Se pueden hacer en tiempo/presupuesto

DOCUMENTO QUE RESPONDA A LAS PREGUNTAS ANTERIORES

II FASE DE DEFINICION.- Hay que contestar

- es factible
- se puede producir
- se puede probar
- vamos a requerir herramienta/recursos especiales
- se necesita un cambio en la organización
- es el presupuesto adecuado
- es económico (costo/beneficio

Esta fase termina cuando se completen los requerimientos y especificaciones del proyecto.

Documento de definición

III DISEÑO.- Esta fase produce un plan para la implementación

ESPECIFICACIONES DISEÑO

- arquitectura del sistema
- tamaño de modulos
- como dividir diseño e implementación
- como se van a probar los modulos y el sistema total
- cual es el perfil de los programadores/analistas
- hay que simular o modelar?
- es factible

Esta fase termina con UN DOCUMENTO en el cual se'deletrean' (especificamente) las especificaciones del diseño y un plan de pruebas

IV FASE DE PROGRAMACION

En esta fase se toman las desiciones técnicas, así como se hace un diseño detallado de programas de computadora
La fase de programación produce modulos codificados que se pueden probar.

El control de calidad del producto empieza en esta fase

Programas fuentes, objeto, notas bibliografia, papers, reportes de problemas, documentos etc. son entregados a la Biblioteca de soporte de programas.

Esta fase termina con la programación (escribir codigo) de todo el sistema integrado y se tiene una 'primera'versión de manuales.

V FASE DE PRUEBA

Esta fase asegura que el producto cumple con los requerimientos funcionales, además verifica que no se tengan errores 'significativos'.
Las pruebas las lleva a cabo un grupo diferente del grupo que programa, y utiliza el documento de 'diseño' y el manual de usuario para llevar a cabo sus pruebas

Todos los errores/solicitudes de cambio se hacen a traves de la Biblioteca.

VI FASE DE ACEPTACION

El usuario (o grupo emulador del usuario) debe aceptar el producto en su forma final.

El "CRITERIO" de aceptación debe ser perfectamente "DEFINIDO" durante la fase de definición, y el cliente debe firmar que esta de acuerdo que el producto que se le va 'entregar es' lo que firma

Las personas que forman este grupo deben de comportarse como el usuario, y llevar a cabo auditorias al producto final.

AQUI TERMINAN LAS FASES DE DESARROLLO

FASES DE EVALUACION

VII INSTALACION

Se inicia cuando el producto sale al "MUNDO REAL" y colocando el producto en las facilidades en las cuales va a ser explotado.

VIII OPERACION

Usualmente el proveedor debe de dar soporte técnico al producto que vende y obtener durante la operación reportes de problemas.

ACTIVIDADES

1.- LA ACTIVIDAD DE ANALISIS (inicio y definición)

El proposito de esta actividad es el desarrollo (y bases) para el diseño.

- se debe entender el problema
- se deben conocer los requisitos
- tener en cuanta factores humanos, ingenieria, documentos herramientas equipos, etc.

Lista de Verificación

( ) Libro de bitacora
( ) Definir las necesidades del usuario (Hay que estar seguro)
( ) Comunicación con el usuario
( ) Conocimeinto de HW existente
( ) Analizar la necesidad de simular/modelar
( ) Investigar proyectos semejantes
( ) Definir y evaluar alternativas
( ) Revisar los documentos de definición con el cliente

2.- PLANEACION

Se debe desarrollar un documento base que sirva de punto de partida entre el cliente y el diseñador.

Calendario

Recursos

Fisicos
Humanos
Economicos

pruebas

criterios de aceptación

plan de implementación

ruta crítica (CPM,PERT)

analisis de riesgos

puntos claves "milestanes"

estimación de costos

Herramientas para el control de calidad

Documentos

standares

desarrollo

programación

diseño

codificación

manuales

Calendario de platicas estructuradas

TABLA PLANEACION

RESUMEN	Tener un resumen del proyecto en general
PLAN	Descripción calendarizada por actividades del proyecto (incluir costos y recursos)
ORGANIZACION	Definición de la organización, y actividades del personal
PRUEBAS	Definir los requerimientos de CC y pruebas de todo el proyecto durante cada fase.

CONTROL DE LA CONFIGURACION	Definir responsabilidades y controles en cada fase y actividad, asi como control en los cambios.
DOCUMENTACION	Definir que documentos se van a elaborar (WP, Estilo, etc)
PLAN DE ENTRENAMIENTO	Definir los requisitos internos y externos de un plan de entrenamiento y los responsables de llevarlo a cabo
REVISIONES, AUDITORIAS Y REPORTES	Definir revisiones en los documentos y productos indicando un avance
PLAN DE INSTALACION Y OPERACION	Describir la forma y requisitos de instalación, asi como un plan para llevarlos a cabo

3.- ACTIVIDAD DE DISEÑO

DISEÑAR BIEN = BUEN PRODUCTO

° Diseño General

° Diseño Detallado

3.1 DISEÑO GENERAL

Las actividades son: Analisis, investigación, estudio, simulación, modelado, diseño de paquetes, uso de herramientas, selección del lenguaje.

HIPOS, CAJAS NEGRAS, D.B. ER.

Lista de verificación de diseño

- Todos los niveles del sistema han sido incorporados
- se han definido las interfases y los modulos
- se han definido las utilerias
- se han hecho consideraciones de tiempo (CPU) espacio memoria etc.
- se han definido las estructuras de datos
- se tiene un procedimiento de implementación, prueba documentación, etc.

DISEÑO DETALLADO

Una vez aprobado el diseño general se lleva a cabo el diseño en el mas fino detalle de cada modelo por ej. en PDL definiendo cada algoritmo, la entrada, etc.

TABLA DISEÑO DETALLADO

- Con calma pero no con flojera
- involucrar a los equipos de desarrollo
- ver si es posible todavía descomponer el sistema (modulo)
- organizar las ideas - escribiendolas

Al final de esta fase de deben tener:

- la producción (lo que se va a producir)
- la descripción detallada de cada unidad
- pruebas de integración
- pruebas del sistema
- parametros de control de calidad
- calendario de laticas estructuradas

UN MANUAL DE LO QUE SE VA A PRODUCIR

ACTIVIDAD DE CODIFICACION

Una vez que se ha diseñado en la forma correcta y apropiada se procede a codificar de arriba hacia abajo. Cada programador debe tener una tarea bien definida del diseño detallado y al terminarlo lo deberá probar en forma unitaria.

- Se deberán tener encabezados detallados de cada modulo
- platicas estructuradas
- c.c.
- un plan detallado de pruebas
- reportes de progreso
- reportes de cambio C.C.
- reportes de control
- reportes de problemas
- manuales cuando menos en cada encabezado

Al final se tienen MODULOS CORRIENDO Y MANUALES

ACTIVIDAD DE PRUEBA

- De acuerdo al plan de pruebas se llevan a cabo estas, por modulos y por sistemas, siempre de arriba hacia abajo, es decir por niveles de jerarquia.

ACTIVIDAD DE ACEPTACION

- DEMO al cliente
- manuales

ACTIVIDAD DE ENTRENAMIENTO

Reglas para las pruebas

- una buena prueba descubre los errores ocultos
- una buena prueba es exhaustiva
- cada prueba debe tener los resultados esperados
- no haga pruebas fuera de programa
- haga pruebas para condiciones invalidas y extremas
- inspeccione los resultados
- la gente que prueba debe de ser la mas creativa
- NO altere los programas para problarlos
- desarrolle datos de prueba
- revise los documentos y manuales

LA NECESIDAD DE LOS DOCUMENTOS

Los documentos resuleven problemas de comunicación

3 tipos de documentos:

TECNICOS.- registros de información de como trabaja el sistema y lo que hace

USUARIO.- registros de que hace el sistema y como usarlo

CONTROL.- información acerca del desarrollo del proyecto.

FASE	DOCUMENTO	TIPO
INICIO	especificaciones de marcado analisis de los requerimientos analisis de factibilidad	TECNICO
DEFINICION	Requerimientos funcionales - descripción de los requerimientos del SW - marco de referencia para el diseño - eficiencia y restricciones - calidad	TECNICO

Plan del proyecto

- plan detallado de trabajo
- costos, tiempos, metodos de control CONTROL.

Cuaderno del proyecto

- cuaderno donde se lleva el control de los eventos y decisiones que afectan el proyecto CONTROL

DISEÑO ESTANDARES DE DISEÑO

- descripción de la arquitectura del WS
- descripción general de los modulos
- interfases y requerimientos de programación TECNICO

+ Plan de C.C.

+ descripción de las metas y pruebas que aseguran un C.C. TECNICO

+ Cambios

+ Control en los cambios y bitacora de cambios CONTROL

+ Manual de Usuarios

+ Documento de como usar el sistema USUARIO

PROGRAMACION

+ Standares de codificación
+ Standares de Diseño
+ Reportes de progreso
+ Reportes de pruebas unitarias
+ Reportes de revisiones

- reportes de cambios

- listados de programas

PRUEBAS + reporte de pruebas

+ reporte de aceptacion TECNICO

PRUEBA DEL SISTEMA

+ STD'S De pruebas

- descripción de las pruebas a llevar a cabo en el sistema TECNICO

OPERACION + Reporte de fallas y problemas

+ Reporte de sugerencias y mejoras

+ Bitacora historica

METODOLOGIAS PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

A N E X O S

Ing. Raymundo H. Rangel Gutiérrez

ABRIL, 1982

Programming Considered as a Human Activity

Introduction

By way of introduction, I should like to start this talk with a story and a quotation.

The story is about the physicist Ludwig Boltzmann, who was willing to reach his goals by lengthy computations. Once somebody complained about the ugliness of his methods, upon which complaint Boltzmann defended his way of working by stating that "elegance was the concern of tailors and shoemakers," implying that he refused to be troubled by it.

In contrast I should like to quote another famous nineteenth century scientist, George Boole. In the middle of his book, *An Investigation of the Laws of Thought*, in a chapter titled "Of the Conditions of a Perfect Method," he writes: "I do not here speak of that perfection only which consists in power, but of that also which is founded in the conception of what is fit and beautiful. It is probable that a careful analysis of this question would conduct us to some such conclusion as the following, viz., that a perfect method should not only be an efficient one, as respects the accomplishment of the objects for which it is designed, but should in all its parts and processes manifest a certain unity and harmony." A difference in attitude one can hardly fail to notice.

Our unconscious association of elegance with luxury may be one of the origins of the not unusual tacit assumption that it costs to be elegant. To show that it also pays to be elegant is one of my prime purposes. It will give us a clearer understanding of the true nature of the quality of programs and the way in which they are expressed, viz., the programming language. From this insight we shall try to derive some clues as to which programming language features are most desirable. Finally, we hope to convince you that the different aims are less conflicting with one another than they might appear to be at first sight.

On the quality of the results

Even under the assumption of flawlessly working machines we should ask ourselves the questions: "When an automatic computer produces results, why do we trust them, if we do so?" and after that: "What measures can we take to increase our confidence that the results produced are indeed the results intended?"

How urgent the first question is might be illustrated by a simple, be it somewhat simplified, example. Suppose that a mathematician interested in number theory has at his disposal a machine with a program to factorize numbers. This process may end in one of two ways: either it gives a factorization of the number given or it answers that the number given is prime. Suppose now that our mathematician wishes to subject to this process a, say, 20 decimal number, while he has strong reasons to suppose that it is a prime number. If the machine confirms this expectation, he will be happy; if it finds a factorization, the mathematician may be disappointed because his intuition has fooled him again, but, when doubtful, he can take a desk machine and can multiply the factors produced in order to check whether the product reproduces the original number. The situation is drastically changed, however, if he expects the number given to be nonprime: if the machine now produces factors he finds his expectations confirmed and moreover he can check the result by multiplying. If, however, the machine comes back with the answer that the number given is, contrary to his expectations and warmest wishes, alas, a prime number, why on earth should he believe this?

Our example shows that even in completely discrete problems the computation of a result is not a well-defined job, well-defined in the sense that one can say: "I have done it," without paying attention to the convincing power of the result, viz., to its *quality*.

The programmer's situation is closely analogous to that of the pure mathematician, who develops a theory and proves results. For a long time pure mathematicians have thought — and some of them still think — that a theorem can be proved completely, that the question whether a supposed proof for a theorem is sufficient or not admits an absolute answer "yes" or "no." But this is an illusion, for as soon as one thinks that one has proved something, one has still the duty to prove that the first proof was flawless, and so on, ad infinitum!

One can never guarantee that a proof is correct; the best one can say "I have not discovered any mistakes." We sometimes flatter ourselves with the idea of giving watertight proofs, but in fact we do nothing but make the correctness of our conclusions plausible. So extremely plausible, that the analogy may serve as a great source of inspiration.

In spite of all its deficiencies, mathematical reasoning presents an outstanding model of how to grasp extremely complicated structures with a brain of limited capacity. And it seems worthwhile to investigate to what extent these proven methods can be transplanted to the art of computer usage. In the design of programming languages one can let oneself be guided primarily by considering "what the machine can do." Considering, however, that the programming language is the bridge between the user and the machine — that it can, in fact, be regarded as his tool — it seems just as important to take into consideration "what Man can think." It is in this vein that we shall continue our investigations.

On the structure of convincing programs

The technique of mastering complexity has been known since ancient times: *Dividi et impera* (Divide and rule). The analogy between proof construction and program construction is, again, striking. In both cases the available starting points are given (axioms and existing theory versus primitives and available library programs); in both cases the goal is given (the theorem to be proved versus the desired performance); in both cases the complexity is tackled by division into parts (lemmas versus subprograms and procedures).

I assume the programmer's genius matches the difficulty of his problem and assume that he has arrived at a suitable subdivision of the task. He then proceeds in the usual manner in the following stages:

- he makes the complete specifications of the individual parts
- he satisfies himself that the total problem is solved provided he had at his disposal program parts meeting the various specifications
- he constructs the individual parts, satisfying the specifications, but independent of one another and the further context in which they will be used.

Obviously, the construction of such an individual part may again be a task of such a complexity, that inside this part of the job, a further subdivision is required.

Some people might think the dissection technique just sketched a rather indirect and tortuous way of reaching one's goals. My own feelings are perhaps best described by saying that I am perfectly aware that there is no Royal Road to Mathematics, in other words, that I have only a very small head and must live with it. I therefore see the dissection technique as one of the rather basic patterns of human understanding and think it worthwhile to try to create circumstances in which it can be most fruitfully applied.

The assumption that the programmer had made a suitable subdivision finds its reflection in the possibility to perform the first two stages: the specification of the parts and the verification that they together do the job. Here elegance, accuracy, clarity and a thorough understanding of the problem at hand are prerequisite. But the whole dissection technique relies on something less outspoken, viz., on what I should like to call "The principle of non-interference." In the second stage above it is assumed that the correct working of the whole can be established by taking, of the parts, into account their exterior specification only, and not the particulars of their interior construction. In the third stage the principle of non-interference pops up again: here it is assumed that the individual parts can be conceived and constructed independently from one another.

This is perhaps the moment to mention that, provided I interpret the signs of current attitudes towards the problems of language definition correctly, in some more formalistic approaches the soundness of the dissection technique is made subject to doubt. Their promoters argue as follows: whenever you give of a mechanism such a two-stage definition, first, what it should do, viz., its specifications, and secondly, how it works, you have, at best, said the same thing twice, but in all probability you have contradicted yourself. And statistically speaking, I am sorry to say, this last remark is a strong point. The only clean way towards language definition, they argue, is by just defining the mechanisms, because what they then will do will follow from this. My question: "How does this follow?" is wisely left unanswered and I am afraid that their neglect of the subtle, but sometimes formidable difference between the concepts *defined* and *known* will make their efforts an intellectual exercise leading into another blind alley.

After this excursion we return to programming itself. Everybody familiar with ALGOL 60 will agree that its procedure concept satisfies to a fair degree our requirements of non-interference, both in its static properties, e.g., in the freedom in the choice of local identifiers, as in its dynamic properties, e.g., the possibility to call a procedure, directly or indirectly, from within itself.

Another striking example of increase of clarity through non-interference, guaranteed by structure, is presented by all programming languages in which algebraic expressions are allowed. Evaluation of such expressions with a sequential machine having an arithmetic unit of limited complexity will imply the use of temporary store for the intermediate results. Their anonymity in the source language guarantees the impossibility that one of them will inadvertently be destroyed before it is used, as would have been possible if the computational process were described in a von Neumann type machine code.

A comparison of some alternatives

A broad comparison between a von Neumann type machine code — well known for its lack of clarity — and different types of algorithmic languages may not be out of order.

In all cases the execution of a program consists of a repeated confrontation of two information streams, the one (say *the program*) constant in time, the other (say *the data*) varying. For many years it has been thought one of the essential virtues of the von Neumann type code that a program could modify its own instructions. In the meantime we have discovered that exactly this facility is to a great extent responsible for the lack of clarity in machine code programs. Simultaneously its indispensability has been questioned: all algebraic compilers I know produce an object program that remains constant during its entire execution phase.

This observation brings us to consider the status of the variable information. Let us first confine our attention to programming languages without assignment statements and without goto statements. Provided that the spectrum of admissible function values is sufficiently broad and the concept of the conditional expression is among the available primitives, one can write the output of every program as the value of a big (recursive) function. For a sequential machine this can be translated into a constant object program, in which at run time a stack is used to keep track of the current hierarchy of calls and the values of the actual parameters supplied at these calls.

Despite its elegance a serious objection can be made against such a programming language. Here the information in the stack can be viewed as objects with nested lifetimes and with a constant value during their entire lifetime. Nowhere (except in the implicit increase of the order counter which embodies the progress of time) is the value of an already existing named object replaced by another value. As a result the only way to store a newly formed result is by putting it on top of the stack; we have no way of expressing that an earlier value now becomes obsolete and the latter's lifetime will be prolonged, although void of interest. Summing up: it is elegant but inadequate. A second objection — which is probably a direct consequence of the first one — is that such programs become after a certain, quickly attained degree of nesting, terribly hard to read.

The usual remedy is the combined introduction of the goto statement and the assignment statement. The goto statement enables us with a backward jump to repeat a piece of program, while the assignment statement can create the necessary difference in status between the successive repetitions.

But I have reasons to ask, whether the goto statement as a remedy is not worse than the defect it aimed to cure. For instance, two programming department managers from different countries and different backgrounds — the one mainly scientific, the other mainly commercial — have communicated to me, independently of each other and on their own initiative, their observation that the quality of their programmers was inversely proportional to the density of goto statements in their programs. This has been an incentive to try to do away with the goto statement.

The idea is, that what we know as *transfer of control*, i.e., replacement of the order counter value, is an operation usually implied as part of more powerful notions: I mention the transition to the next statement, the procedure call and return, the conditional clauses and the for statement; and it is the question whether the programmer is not rather led astray by giving him separate control over it.

I have done various programming experiments and compared the ALGOL text with the text I got in modified versions of ALGOL 60 in which the goto statement was abolished and the for statement — being pompous and overelaborate — was replaced by a primitive repetition clause. The latter versions were more difficult to make; we are so familiar with the jump order that it requires some effort to forget it. In all cases tried, however, the program without the goto statements turned out to be shorter and more lucid.

The origin of the increase in clarity is quite understandable. As is well known there exists no algorithm to decide whether a given program ends or not. In other words, each programmer who wants to produce a flawless program must at least convince himself by inspection that his program will indeed terminate. In a program in which unrestricted use of the goto statement has been made, this analysis may be very hard on account of the great variety of ways in which the program may fail to stop. After the abolishment of the goto statement there are only two ways in which a program may fail to stop: either by infinite recursion, i.e., through the procedure mechanism, or by the repetition clause. This simplifies the inspection greatly.

The notion of repetition, so fundamental in programming, has a further consequence. It is not unusual that inside a sequence of statements to be repeated one or more subexpressions occur, which do not change their value during the repetition. If such a sequence is to be repeated many times, it would be a regrettable waste of time if the machine had to recompute these same values over and over again. One way out of this is to delegate to the now optimizing translator the discovery of such constant sub-expressions in order that it can take the computation of their values outside the loop. Without an optimizing translator the obvious solution is to invite the programmer to be somewhat more explicit and he can do so by introducing as many additional variables as there are constant sub-expressions within the repetition and by assigning the values to them before entering the repetition. I should like to stress that both ways of writing the program are equally misleading. In the first case the translator is faced with the unnecessary puzzle to discover the constancy; in the second case we have introduced a variable, the only function of which is to denote a constant value. This last observation shows the way out of the

difficulty: besides variables the programmer would be served by *local constants*, i.e., identifiable quantities with a finite lifetime, during which they will have a constant value, that has been defined at the moment of introduction of the quantity. Such quantities are not new: the formal parameters of procedures already display this property. The above is a plea to recognize that the concept of the *local constant* has its own right of existence. If I am well informed, this has already been recognized in CPL, the programming language designed in a joint effort around the Mathematical Laboratory of the University of Cambridge, England.

The double gain of clarity

I have discussed at length that the convincing power of the results is greatly dependent on the clarity of the program: on the degree in which it reflects the structure of the process to be performed. For those who feel themselves mostly concerned with efficiency as measured in the cruder units of storage space and machine time: I should like to point out that increase of efficiency always comes down to exploitation of structure and for them I should like to stress that all structural properties mentioned can be used to increase the efficiency of an implementation. I shall review them briefly.

The lifetime relation satisfied by the local quantities of procedures allows us to allocate them in a stack, thus making very efficient use of available store; the anonymity of the intermediate results enables us to minimize storage references dynamically with the aid of an automatically controlled set of push down accumulators; the constancy of program text under execution is of great help in machines with different storage levels and reduces the complexity of advanced control considerably; the repetition clause eases the dynamic detection of endless looping and finally, the local constant is a successful candidate for a write-slow-read-fast store, when available.

Conclusion

When I became acquainted with the notion of algorithmic languages I never challenged the then prevailing opinion that the problems of language design and implementation were mostly a question of compromises: every new convenience for the user had to be paid for by the implementation, either in the form of increased trouble during translation, or during execution or during both. Well, we are most certainly not living in Heaven and I am not going to deny the possibility of a conflict between convenience and efficiency, but now I do protest when this conflict is presented as a complete summing up of the situation. I am of the opinion that it is worthwhile to investigate to what extent the needs of Man and Machine go hand in hand and to see what techniques we can devise for the benefit of all of us. I trust that this investigation will bear fruits and if this talk made some of you share this fervent hope, it has achieved its aim.

Go To Statement Considered Harmful

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to

shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of go to statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (*if B then A*), alternative clauses (*if B then A1 else A2*), choice clauses as introduced by C.A.R. Hoare (*case [i] of [A1, A2, . . . , An]*), or conditional expressions as introduced by J. McCarthy (*B1 → E1, B2 → E2, . . . , Bn → En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, while *B* repeat *A* or repeat *A* until *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is — and this seems to be inherent to sequential processes — that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the go to statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the go to statement is far from new. I remember having read the explicit recommendation to restrict the use of the go to statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C.A.R. Hoare. In [1, Sec. 3.2.1.] Wirth and

Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than go to statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References

1. N. Wirth and C.A.R. Hoare, "A Contribution to the Development of ALGOL," *Communications of the ACM*, Vol. 9, No. 6 (June 1966), pp 413-32.
2. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.

Structured Programming

Introduction

This working document reports on experience and insights gained in programming experiments performed by the author in the last year. The leading question was if it was conceivable to increase our programming ability by an order of magnitude and what techniques (mental, organizational or mechanical) could be applied in the process of program composition to produce this increase. The programming experiments were undertaken to shed light upon these questions.

Program size

My real concern is with intrinsically large programs. By "intrinsically large" I mean programs that are large due to the complexity of their task, in contrast to programs that have exploded (by inadequacy of the equipment, unhappy decisions, poor understanding of the problem, etc.). The fact that, for practical reasons, my experiments had thus far to be carried out with rather small programs did present a serious difficulty; I have tried to overcome this by treating problems of size explicitly and by trying to find their consequences as much as possible by analysis, inspection and reflection, rather than by (as yet too expensive) experiments.

In doing so I found a number of subgoals that, apparently, we have to learn to achieve (if we don't already know how to do that).

If a large program is a composition of N "program components," the confidence level of the individual components must be exceptionally high if N is very large. If the individual components can be made with the probability "p" of being correct, the probability that the whole program functions properly will not exceed

$$P = p^N$$

for large N, p must be practically equal to one if P is to differ significantly from zero. Combining subsets into larger components from which then the whole program is composed, presents no remedy:

$$p^{N/2} \cdot p^{N/2} \text{ still equals } p^N$$

As a consequence, the problem of program correctness (confidence level) was one of my primary concerns.

The effort — be it intellectual or experimental — needed to demonstrate the correctness of a program in a sufficiently convincing manner may (measured in some loose sense) not grow more rapidly than in proportion to the program length (measured in an equally loose sense). If, for instance, the labour involved in verifying the correct composition of a whole program out of N program components (each of them individually assumed to be correct) still grows exponentially with N, we had better admit defeat.

Any large program will exist during its life-time in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program, but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program therefore should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstration of the shared components but also of the shared substructure.

Program correctness

An assertion of program correctness is an assertion about the net effects of the computations that may be evoked by this program. Investigating how such assertions can be justified, I came to the following conclusions:

1. The number of different inputs, i.e. the number of different computations for which the assertions claim to hold is so fantastically high that demonstration of correctness by sampling is completely out of the question. *Program testing can be used to show the presence of bugs, but never to show their absence!*

   Therefore, proof of program correctness should depend only upon the program text.

2. A number of people have shown that program correctness can be proved. Highly formal correctness proofs have been given; also correctness proofs have been given for "normal programs," i.e. programs not written with a proof procedure in mind. As is to be expected (and nobody is to be blamed for that) the circulating examples are concerned with rather small programs and, unless measures are taken, the amount of labour involved in proving might well (will) explode with program size.

3. Therefore, I have not focused my attention on the question "how do we prove the correctness of a given program?" but on the questions "for what program structures can we give correctness proofs without undue labour, even if the programs get large?" and, as a sequel, "how do we make, for a given task, such a well-structured program?" My willingness to confine my attention to such "well-structured programs" (as a subset of the set of all possible programs) is based on my belief that we can find such a well-structured subset satisfying our programming needs, i.e. that for each programmable task this subset contains enough realistic programs.

4. This, what I call "constructive approach to the problem of program correctness," can be taken a step further. It is not restricted to general considerations as to what program structures are attractive from the point of view of provability, in a number of specific, very difficult programming tasks I have finally succeeded in constructing a program by analyzing how a proof could be given that a class of computations would satisfy certain requirements; from the requirements of the proof the program followed.

The relation between program and computation

Investigating how assertions about the possible computations (evolving in time) can be made on account of the static program text, I have concluded that adherence to rigid sequencing disciplines is essential, so as to allow step-wise abstraction from the possibly different routings. In particular: when programs for a sequential computer are expressed as a linear sequence of basic symbols of a programming language, sequencing should be controlled by alternative, conditional and repetitive clauses and procedure calls, rather than by statements transferring control to labelled points.

The need for step-wise abstraction from local sequencing is perhaps most convincingly shown by the following demonstration:

Let us consider a "stretched" program of the form

$$S_1; S_2; \ldots; S_N \qquad (1)$$

and let us introduce the measuring convention that when the net effect of the execution of each individual statement S_i has been given, it takes N steps of reasoning to establish the correctness of program (1), i.e. to establish that the cumulative net effect of the N actions in succession satisfies the requirements imposed upon the computations evoked by program (1).

For a statement of the form

$$\text{if } B \text{ then } S_1 \text{ else } S_2 \qquad (2)$$

where, again, the net effect of the execution of the constituent statements S_1 and S_2 has been given; we introduce the measuring convention that it takes 2 steps of reasoning to establish the net effect of program (2); viz. one for the case B and one for the case not B.

Consider now a program of the form

$$\begin{array}{l} \text{if } B_1 \text{ then } S_{11} \text{ else } S_{12}; \\ \text{if } B_2 \text{ then } S_{21} \text{ else } S_{22}; \\ \qquad \ldots \\ \text{if } B_N \text{ then } S_{N1} \text{ else } S_{N2} \end{array} \qquad (3)$$

According to the measuring convention it takes 2 steps per alternative statement to understand it, i.e. to establish that the net effect of

$$\text{if } B_i \text{ then } S_{i1} \text{ else } S_{i2}$$

is equivalent to that of the execution of an abstract statement S_i. Having N such alternative statements, it takes us 2N steps to reduce program (3) to one of the form of program (1); to understand the latter form of the program takes us another N steps, giving 3N steps in toto.

If we had refused to introduce the abstract statements S_i but had tried to understand program (3) directly in terms of executions of the statements S_{ij}, each such computation would be the cumulative effect of N such statement executions and would as such require N steps to understand it. Trying to understand the algorithm in terms of the S_{ij} implies that we have to distinguish between 2^N different routings through the program and this would lead to $N*2^N$ steps of reasoning!

I trust that the above calculation convincingly demonstrates the need for the introduction of the abstract statements S_i. An aspect of my constructive approach is not to reduce a given program (3) to an abstract program (1), but to start with the latter.

Abstract data structures

Understanding a program composed from a modest number of abstract statements again becomes an exploding task if the definition of the net effect of the constituent statements is sufficiently unwieldy. This can be overcome by the introduction of suitable abstract data structures. The situation is greatly analogous to the way in which we can understand an ALGOL program operating on integers without having to bother about the number representation of the implementation used. The only difference is that now the programmer must invent his own concepts (analogous to the "ready-made" integer) and his own operations upon them (analogous to the "ready-made" arithmetic operations).

In the refinement of an abstract program (i.e. composed from abstract statements operating on abstract data structures) we observe the phenomenon of "joint refinement." For abstract data structures of a given type a certain representation is chosen in terms of new (perhaps still rather abstract) data structures. The immediate consequence of this design decision is that the abstract statements operating upon the original abstract data structure have to be redefined in terms of algorithmic refinements operating upon the new data structures in terms of which it was decided to represent the original abstract data structure. Such a joint refinement of data structure and associated statements should be an isolated unit of the program text: it embodies the immediate consequences of an (independent) design decision and is as such the natural unit of interchange for program modification. It is an example of what I have grown into calling "a pearl."

Programs as necklaces strung from pearls

I have grown to regard a program as an ordered set of pearls; a "necklace." The top pearl describes the program in its most abstract form, in all lower pearls one or more concepts used above are explained (refined) in terms of concepts to be explained (refined) in pearls below it, while the bottom pearl eventually explains what still has to be explained in terms of a standard interface (=machine). The pearl seems to be a natural program module.

As each pearl embodies a specific design decision (or, as the case may be, a specific aspect of the original problem statement) it is the natural unit of interchange in program modification (or, as the case may be, program adaptation to a change in problem statement).

Pearls and necklace give a clear status to an "incomplete program," consisting of the top half of a necklace; it can be regarded as a complete program to be executed by a suitable machine (of which the bottom half of the necklace gives a feasible implementation). As such, the correctness of the upper half of the necklace can be established regardless of the choice of the bottom half.

Between two successive pearls we can make a "cut," which is a manual for a machine provided by the part of the necklace below the cut and used by the program represented by the part of the necklace above the cut. This manual serves as an interface between the two parts of the necklace. We feel this form of interface more helpful than regarding data representation as an interface between operations, in particular more helpful towards ensuring the combinatorial freedom required for program adaptation.

The combinatorial freedom just mentioned seems to be the only way in which we can make a program as part of a family or "in many (potential) versions" without the labour involved increasing proportional to the number of members of the family. The family becomes the set of those selections from a given collection of pearls that can be strung into a fitting necklace.

Concluding remarks

Pearls in a necklace have a strict logical order, say "from top to bottom." I would like to stress that this order may be radically different from the order (in time) in which they are designed.

Pearls have emerged as program modules when I tried to map upon each other as completely as possible, the numerous members of a class of related programs. The abstraction process involved in this mapping turns out (not, amazingly, as an afterthought!) to be the same as the one that can be used to reduce the amount of intellectual labour involved in correctness proofs. This is very encouraging.

As I said before, the programming experiments have been carried out with relatively small programs. Although, personally, I firmly believe that they show the way towards more reliable composition of really large programs, I should like to stress that as yet I have *no* experimental evidence for this. The experimental evidence gained so far shows an increasing ability to compose programs of the size I tried. Although I tried to do it, I feel that I have given but little recognition to the requirements of program development such as is needed when one wishes to employ a large crowd; I have no experience with the Chinese Army approach, nor am I convinced of its virtues.

The Translation of 'go to' Programs to 'while' Programs

1. GENERAL DISCUSSION

1.1. Introduction

The first class of programs we consider are simple *flowchart programs* constructed from assignment statements (that assign terms to variables) and test statements (that test quantifier-free formulas) operating on a "state vector" $\bar{x}$. The flowchart program begins with a unique start statement of the form

where $\bar{x}_{input}$ is a subvector of $\bar{x}$ indicating the variables that have to be given values at the beginning of the computation. It ends with a unique halt statement of the form

where $\bar{x}_{output}$ is a subvector of $\bar{x}$ indicating the variables whose values will be the desired result of the computation.

We make no assumptions about the domain of individuals, or about the operations and predicates used in the statements. Thus our flowchart programs are really flowchart schemas (see, for example, Luckham, Park and Paterson [1]) and all the results can be stated in terms of such schemas.

Let P_1 be any flowchart program of the form shown in Figure 1. Note that, for example, the statement $\bar{x} \leftarrow e(\bar{x})$ stands for any sequence of assignment statements whose net effect is the replacement of vector $\bar{x}$ by a new vector $e(\bar{x})$. Similarly, the test $p(\bar{x})$, for example, stands for any quantifier-free formula with variables from $\bar{x}$. The flowchart program P_1 will be used as an example throughout the paper

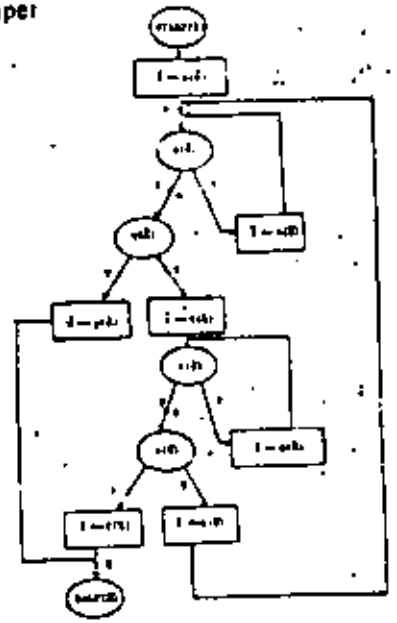


Figure 1. The flowchart program P_1.

In order to write such flowchart programs in a conventional programming language, *goto* statements are required. There has recently been much discussion (see, for example, Dijkstra [2]) about whether the use of *goto* statements makes programs difficult to understand, and whether the use of *while* or *for* statements is preferable. In addition, it is quite possible that simpler proof methods of the validity and equivalence of programs may be found for programs without *goto* statements (see, for example, Stark [3]). It is clearly relevant to this discussion to consider whether the abolition of *goto* statements is really possible.

Therefore the second class of programs we consider are *while programs*, i.e., Algol-like programs consisting only of *while* statements of the form *while* (quantifier-free formula) *do* (statement), in addition to conditional, assignment and block* statements. As before, each program starts with a unique start statement, START($\bar{x}_{input}$), and ends with a unique halt statement, HALT($\bar{x}_{output}$).

*A block statement is denoted by any sequence of statements enclosed by square brackets

Since both classes of programs use the same kind of start and halt statements, we can define the equivalence of two programs independently of the classes to which they belong. Two programs (with the same input subvectors $\bar{x}_{input}$ and the same output subvectors $\bar{x}_{output}$) are said to be *equivalent* if for each assignment of values to $\bar{x}_{input}$ either both programs do not terminate or both terminate with the same values in $\bar{x}_{output}$.

1.2. Translation to while programs by adding variables

1.2.1. Extending the state vector $\bar{x}$

We can show that by allowing extra variables which keep crucial past values of some of the variables in $\bar{x}$, one can effectively translate every flowchart program into an equivalent while program (ALGORITHM I). The importance of this result is that original "topology" of the program is preserved, and the new program is of the same order of efficiency as the original program. However, we shall not enter into any discussion as to whether the new program is superior to the original one or not.

This result, considered in terms of schemas, can be contrasted with those of Paterson and Hewitt [4] (see also Strong [5]). They showed that although it is not possible to translate all recursive schemas into flowchart schemas, it is possible to do this for "linear" recursive schemas, by adding extra variables. However, as they point out, the flowchart schemas produced are less efficient than the original recursive schemas.

As an example, ALGORITHM I will give the following while program which is equivalent to the flowchart program P_1 (Figure 1):

START($\bar{x}$);
$\bar{x} \leftarrow a(\bar{x})$;
[*while* $p(\bar{x})$ *do* $\bar{x} \leftarrow e(\bar{x})$;
$\bar{y} \leftarrow \bar{x}$;
if $q(\bar{x})$ *then* [$\bar{x} \leftarrow b(\bar{x})$; *while* $r(\bar{x})$ *do* $\bar{x} \leftarrow d(\bar{x})$]];
while $q(\bar{y}) \wedge s(\bar{x})$ *do*
  [$\bar{x} \leftarrow c(\bar{x})$;
  [*while* $p(\bar{x})$ *do* $\bar{x} \leftarrow e(\bar{x})$;
  $\bar{y} \leftarrow \bar{x}$;
  if $q(\bar{x})$ *then*
  [$\bar{x} \leftarrow b(\bar{x})$; *while* $r(\bar{x})$ *do* $\bar{x} \leftarrow d(\bar{x})$]]];
if $q(\bar{y})$ *then* $\bar{x} \leftarrow f(\bar{x})$ *else* $\bar{x} \leftarrow g(\bar{x})$;
HALT($\bar{x}$).

If the test $q(\bar{x})$ uses only a subvector of $\bar{x}$, then the algorithm will indicate that the vector of extra variables $\bar{y}$ need only be of the same length as this subvector.

Note that on each cycle of the main while statement, the state vector $\bar{x}$ is at point β, while $\bar{y}$ holds the preceding values of $\bar{x}$ at point α.

Note also that the two subprograms enclosed in broken lines are identical. This is typical of the programs produced by the algorithm. One might use this fact to make the programs more readable by using "subroutines" for the repeated subprograms.

Because of space limitations we cannot present ALGORITHM I in this paper. The detailed algorithm can be found in the preliminary report of this paper (CS 188, Computer Science Dept., Stanford University).

1.2.2. *Adding boolean variables*

The translation of flowchart programs into while programs by the addition of boolean variables is not a new idea. Böhm and Jacopini [6] and Cooper [7] (see also Bruno and Steiglitz [8]) have shown that every flowchart program can be effectively translated into an equivalent while program (with one while statement) by introducing new boolean variables into the program, new predicates to test these variables, together with assignments to set them *true* or *false*. The boolean variables essentially simulate a program counter, and the while program simply interprets the original program. On each repetition of the while statement, the next operation of the original program is performed, and the "program counter" is updated. As noted by Cooper and Bruno and Steiglitz themselves, this transformation is undesirable since it changes the "topology" (loop-structure) of the program, giving a program that is less easy to understand. For example, if a while program is written as a flowchart program and then transformed back to an equivalent while program by their method, the resulting while program will not resemble the original.

We give an algorithm (ALGORITHM II) for transforming flowchart programs to equivalent while programs by adding extra boolean variables, which is an improvement on the above method. It preserves the "topology" of the original program and in particular it does not alter while-like structure that may already exist in the original program.

For the flowchart program P_1 (Figure 1), for example ALGORITHM II will produce the following while program.

```
START(x̄);
x̄ ← α(x̄);
t ← true;
while t do
    [while p(x̄) do x̄ ← c(x̄);
     if q(x̄) then [x̄ ← b(x̄);
         while r(x̄) do x̄ ← d(x̄);
         if s(x̄) then x̄ ← e(x̄)
             else [x̄ ← f(x̄); t ← false]]
         else [x̄ ← g(x̄); t ← false]];
HALT(x̄).
```

Note that each repetition of the main while statement starts from point γ and proceeds either back to γ or to δ. In the latter case, t is made *false* and we subsequently exit from the while statement.

1.3. Translation to while programs without adding variables

It is natural at this point to consider whether every flowchart program can be translated into an equivalent while program without adding extra variables (i.e., using only the original state vector $\bar{x}$). We show that this cannot be done in general, and in fact there is a flowchart program of the form of Figure 1 which is an appropriate counter-example.

A similar negative result has been demonstrated by Knuth and Floyd [9] and Scott [private communication]. However, the notion of equivalence considered by those authors is more restrictive in that it requires equivalence of computation sequences (i.e., the sequence of assignment and test statements in order of execution) and not just the equivalence of final results of computation as we do. Thus, since our notion of equivalence is weaker, our negative result is stronger.

2. ALGORITHM II: TRANSLATION BY ADDING BOOLEAN VARIABLES

The second algorithm, ALGORITHM II, translates flowchart programs to equivalent while programs by adding boolean variables. It makes use of the fact that every flowchart program (without the start and halt statements) can be decomposed into blocks where a block is any piece of flowchart program with only one exit (but possibly many entrances). This is obvious since in particular the whole body of the given flowchart program can be considered as such a block. The aim, whenever possible, is to get blocks containing at most one top-level test statement (i.e., test statement not contained in inner blocks) since such blocks can be represented as a piece of while program without adding boolean variables. In particular, if a while program is expressed as a flowchart program, this latter program can always be decomposed into such simple blocks, and the algorithm will give us back the original while program.

For any given flowchart program we construct the equivalent while program by induction on the structure of the blocks.

For each entrance b_i to block B we consider that part B_i of the block reachable from b_i. We then recursively construct an equivalent piece of while program $\bar{B}_i(\bar{x}, \bar{t})^*$ as follows. There are two cases to consider:

Case 1: (a) B_i contains at most one top-level test statement.
or (b) B_i contains no top-level loops.

*$\bar{t}$ is a (possibly empty) vector of additional boolean variables introduced by the translation.

Let us assume that we have a while program P'_2 equivalent to P_2 which also has one variable and the same domain D. Although we could allow the assignment statements of P'_2 to use any terms obtained by compositions of the operations G and H, we assume without loss of generality that each assignment statement in P'_2 consists of a single operation G or H. The tests in the conditional and while statements may only use quantifier-free formulas obtained from tests p and q, and operations G and H. Since we use only one variable, it follows that the sequences of values describing corresponding computations of P'_2 and P_2 are identical. Note also that since there is a bound on the depth of terms in the quantifier-free formulas, there is a bound, M say, on the number of leftmost letters in the tail that can affect the decision of any test in P'_2. Finally, without loss of generality we shall make the restriction that there is no redundant while statement in P'_2; i.e., there is no while statement with a uniform bound on the number of its iterations in terminating computations.

Since P'_2 must contain some (non-redundant) while statement, let W be any while statement in P'_2 which is not contained or followed by another while statement. The point in P'_2 immediately after W we shall denote by A.

Lemma: For all n ($n \geq 0$) there exist strings $a, c \in [\alpha, \beta]^*$ and $d \in [\alpha, \beta]^{**}$ ($|c| = n$)[1] such that for all strings $b \in [\alpha, \beta]^*$ the computation of P'_2 starting with tail $abcyd$ passes A with some tail $\underline{ab}cyd$, where $\underline{ab}$ is some rightmost substring of ab (possibly empty).

From this lemma we immediately obtain the following corollary.

Corollary. For every n, $n \geq 0$, there exists a finite computation of P'_2 which passes through A with more than n operations still to be performed.

But this contradicts the fact that since there is no while statement following A, the number of operations that P'_2 can perform after A is bounded.

Proof of Lemma: By induction on n.

Base step. Choose any computation starting with tail $a'a''b'\gamma d'$ ($a', a'', b' \in [\alpha, \beta]^*$, $d' \in [\alpha, \beta]^{**}$ and $|a''| = M$) that enters W with tail $a''b'\gamma d'$. (Such computation exists by non-redundancy of W.)

Since at most M leftmost letters of the tail can effect the decision of any test, on entering W the main test can only look at a''. Therefore the test will be true for any tail starting with a''.

In particular, the computation starting with tail $a'a''b\gamma a''d'$, *for any* $b \in [\alpha, \beta]^*$, also enters W at the same point, i.e., with tail $a''b\gamma a''d'$. Since the computation is finite, it must subsequently pass point A, but (noting that the test in W must be false when passing A) it cannot pass A with tail $a''d'$.

Hence, with $a = a'a''$, $d = a''d'$, for all strings b in $[\alpha, \beta]^*$, the computation starting with $ab\gamma d$ must pass A with some tail $\underline{ab}\gamma d$ where $\underline{ab}$ is some rightmost substring of ab.

Induction step. Assume we have strings $a, c \in [\alpha, \beta]^*$ and $d \in [\alpha, \beta]^{**}$, $|c| = n$, such that for all strings b in $[\alpha, \beta]^*$ the computation starting with tail $abc\gamma d$ passes A with some tail $\underline{ab}c\gamma d$ where $\underline{ab}$ is some rightmost substring of ab.

We find a string $c' \in [\alpha, \beta]^*$, $|c'| = n+1$, such that for all strings b' in $[\alpha, \beta]^*$ the computation starting with tail $ab'c'\gamma d$ passes A with some tail $\underline{ab'}c'\gamma d$ where ab' is some rightmost substring of $\underline{ab'}$.

There are three cases to consider:

(i) For all nonempty strings b, the corresponding substring $\underline{ab}$ is nonempty. In this case *we take* c' *to be* αc.[2]

For any string b' in $[\alpha, \beta]^*$ the computation starting with tail $ab'\alpha c\gamma d$, passes A with tail $\underline{ab'}\alpha c\gamma d$, where $\underline{ab'}$ is a rightmost substring of ab'.

(ii) For some nonempty string $b = b''\alpha$ ($b'' \in [\alpha, \beta]^*$), the substring $\underline{ab}$ is empty, i.e., there exists computation S starting with $ab''\alpha c\gamma d$ that passes A with $c\gamma d$. In this case *we take* c' *to be* βc.

By earlier remarks about P_1 and P'_2, it follows that the next operation in S after passing A must be H.

Now, for *any string* b' *in* $[\alpha, \beta]^*$ the computation starting with tail $ab'\beta c\gamma d$ must pass A with some tail $\underline{ab'\beta}c\gamma d$ where $\underline{ab'\beta}$ is some rightmost substring of $ab'\beta$.

$\underline{ab'\beta}$ cannot be empty because this would mean that this computation passes A with the same tail $c\gamma d$ as for S but in this case the next operation to be performed is G. This is impossible, since the course of computation from A must be determined by the tail at this point.

[1] I.e., a and c are finite strings (possibly empty) over $[\alpha, \beta]$, d is an infinite string over $[\alpha, \beta]$ and the length of c is n.

[2] We could equally well take c' to be βc and consider computations starting with tail $ab'\beta c\gamma d$.

Hence, the computation must pass A with some tail $\underline{ab'}\beta c\gamma d$ (i.e., $\underline{ab'}c'\gamma d$) where ab' is a rightmost substring of ab'.

(iii) For some nonempty string $b = b''\beta\ (b'' \in [\alpha, \beta]^+)$, the substring $\underline{ab}$ is empty. In this case we take c' to be αc.

We proceed as in case (ii) with α and β interchanged and G and H interchanged.

Acknowledgement

We are indebted to David Cooper for stimulating discussions and mainly for his idea of using cut-set points which we have adopted in ALGORITHM II. We are also grateful to Donald Knuth for his critical reading of the manuscript and subsequent helpful suggestions.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

References

1. D.C. Luckham, D.M.R. Park, and M.S. Paterson, "On Formalized Computer Programs," *Journal of Computer and System Sciences*, Vol. 4, No. 3 (June 1970), pp. 220-49.
2. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.
3. R. Stark, "A Language for Algorithms," *Computer Journal*, Vol. 14, No. 1 (February 1971), pp. 40-44.
4. M.S. Paterson and C.E. Hewitt, "Comparative Schematology," unpublished memo.
5. H.R. Strong, "Translating Recursion Equations into Flowcharts," *Journal of Computer and System Sciences*, Vol. 5, No. 3 (June 1971), pp. 254-85.
6. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.
7. D.C. Cooper, "Böhm and Jacopini's Reduction of Flowcharts," *Communications of the ACM*, Vol. 10, No. 8 (August 1967), pp. 463-73.
8. J. Bruno and K. Steiglitz, "The Expression of Algorithms by Charts," *Journal of the ACM*, Vol. 19, No. 3 (July 1972), pp. 517-25.
9. D.E. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31; see also Stanford University Computer Science Technical Report, Vol. CS148 (Stanford, Calif.: January 1970).

WULF 8

A Case Against the GOTO

Introduction

It has been suggested that the use of the goto construct is undesirable, is bad programming practice, and that at least one measure of the "quality" of a program is inversely related to the number of goto statements contained in it. The rationale behind this suggestion is that it is possible to use the goto in ways which obscure the logical structure of a program, thus making it difficult to understand, modify, debug, and/or prove its correctness. It is quite clear that not all uses of the goto are obscure, but the hypothesis is that these situations fall into one of a small number of cases and therefore explicit and inherently well-structured language constructs may be introduced to handle them. Although the suggestion to ban the goto appears to have been a part of the computing folklore for several years, to this author's knowledge the suggestion was first made in print by Professor E.W. Dijkstra in a letter to the editor of the *Communications of the ACM* in 1968 [1].

In this paper we shall examine the rationale for the elimination of the goto in programming languages, and some of the theoretical and practical implications of its (total) elimination.

Rationale

At one level, the rationale for eliminating the goto has already been given in the introduction. Namely, it is possible to use the goto in a manner which obscures the logical structure of a program to a point where it becomes virtually impossible to understand [1, 3, 4]. It is *not* claimed that *every* use of the goto obscures the logical structure of a program; it is only claimed that it is *possible* to use the goto to fabricate a "rat's nest" of control flow which has the undesirable properties mentioned above. Hence this argument addresses the *use* of the goto rather than the goto itself.

As the basis for a proposal to totally eliminate the goto this argument is somewhat weak. It might reasonably be argued that the undesirable consequences of unrestricted branching may be eliminated by enforcing restrictions on the *use* of the goto rather than eliminating the construct. However, it will be seen that any rational set of restrictions is equivalent to eliminating the construct if an adequate set of other control primitives is provided. The strong reasons *for* eliminating the goto arise in the context of more positive proposals for a programming methodology which makes the goto unnecessary. It is not the purpose of this paper to explicate these methodologies (variously called "structured programming," "constructive programming," "stepwise refinement," etc.); however, since the major justification for eliminating the goto lies in this work, a few words are in order.

It is, perhaps, pedantic to observe that the present practice of building large programming systems is a mess. Most, if not all, of the major operating systems, compilers, information systems, etc. developed in the last decade have been delivered late, have performed below expectation (at least initially), and have been filled with "bugs." This situation is intolerable, and has prompted several researchers [2, 3, 4, 5, 6, 7, 8, 9] to consider whether a programming methodology might be developed to correct this situation. This work has proceeded from two premises:

1. Dijkstra speaks of our "human inability to do much" (at one time) to point up the necessity of decomposing large systems into smaller, more "human size" chunks. This observation is hardly startling, and in fact, most programming languages include features (modules, subroutines, and macros, for example) to aid in the mechanical aspects of this decomposition. However, the further observation that the particular decomposition chosen makes a significant difference to the understandability, modifiability, etc., of a program and that there is an *a priori* methodology for choosing a "good" decomposition is less expected.

2. Dijkstra has also said that debugging can show the presence of errors, but never their absence. Thus ultimately we will have to be able to prove the correctness of the programs we construct (rather than "debug" them) since their sheer size prohibits exhaustive testing. Although some progress has been made on the automatic proof of the correctness of programs (c.f., [10, 11, 12, 23, 24]), this approach appears to be far from a practical reality. The methodology proposed by Dijkstra (and others) proceeds so that the construction of a program guides a (comparatively) simple and intuitive proof of its correctness.

The methodology of "constructive programming" is quite simple and, in this context, best described by an (partial) example. Let us consider the problem of producing a KWIC* index. Construction of the program proceeds in a series of steps in which each step is a refinement of some portion of a previous step. We start with a single statement of the function to be performed:

```
Step 1:  PAINTKWIC
```

We may think of this as being an instruction in a language (or machine) in which the notion of generating a KWIC index is primitive. Since this operation is not primitive in most practical languages, we proceed to define it:

```
Step 2:  PAINTKWIC:  generate and save all interesting circular shifts
                     alphabetize the saved lines
                     print alphabetized lines
```

Again, we may think of each of these lines as being an instruction in an appropriate language; and again, since they are not primitive in most existing languages, we must define them; for example:

```
Step 3a:  generate and save all interesting circular shifts:
                  for each line in the input do
                     begin
                     generate and save all interesting
                        shifts of "this line"
                     end
          etc.
```

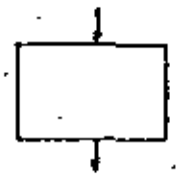
*For those who may not be familiar with a KWIC (key word in context) index, the following description is adequate for this paper.

A KWIC system accepts a set of *lines*. Each *line* is an ordered set of *words* and each *word* is an ordered set of characters. A *word* may be one of a set of *uninteresting words* ("a," "the," "of," etc.), otherwise it is an *interesting word*. Any line may be circularly shifted by removing its first word and placing it at the end of the line. The KWIC index system generates an ordered (alphabetically by the first word) listing of all circular shifts of all the input lines such that no line in the output begins with an *uninteresting word*.

The construction of the program proceeds by small steps* in this way until ultimately each operation is expressed in the available primitive operations of the target language. We shall not carry out the details since the objective of this paper is not to be a tutorial on this methodology. However, note that the methodology achieves the goals set out for it. Since the context is small at each step it is relatively easy to understand what is going on; indeed, it is easy to prove that the program will work correctly if the primitives from which it is constructed are correct. Moreover, proving the correctness of the primitives used at step i is a small set of proofs (of the same kind) at step $i+1$. (In the terminology of this methodology, step i is an *abstraction* from its implementation in step $i+1$.)

Now, the constructive programming methodology relates to eliminating the goto in the following way. It is crucial to the constructive philosophy that it should be possible to define the behavior of each primitive action at the ith step independent of the context in which it occurs. If this were not so, it would not be possible to prove the correctness of these primitives at the $i+1$st step without reference to their context in the ith step. In particular, this suggests (using flow chart terminology) that it should be possible to represent each primitive at the ith step by a (sub) flow chart with a single entry and a single exit path. Since this must be true at each step of the construction, the final flow chart of a program constructed in this way must consist of a set of totally nested (sub)-flow charts. Such a flow chart can be constructed without an explicit goto if conditional and looping constructs are available.

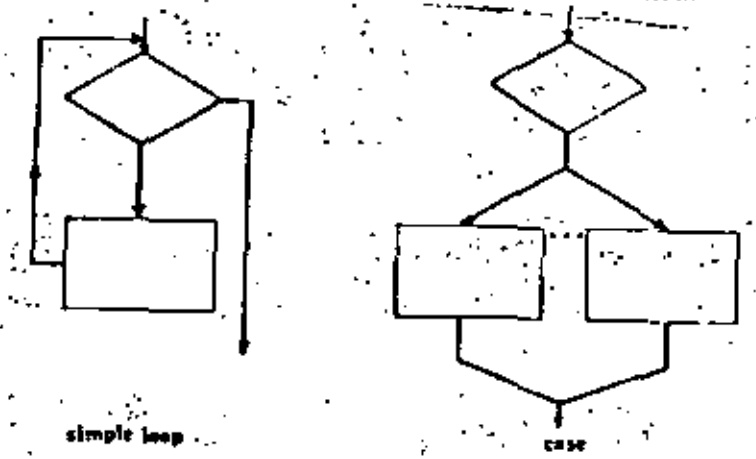
Consider, now, programs which can be built from only simple conditional and loop constructs. To do this we will use a flow chart representation because of the explicit way in which it manifests control. We assume two basic flow chart elements, a "process" box and a "binary decision" box:



These boxes are connected by directed line segments in the usual way. We are interested in two special "goto-less" constructions fabricated from these primitives: a simple loop and an n-way conditional, or "case," construct. We consider these forms "goto-less" since they contain single entry and exit points and hence might reasonably be provided in a language by explicit syntactic constructs. (The loop considered here obviously does not correspond to all vari-

*A more complete explication of the methodology would concern itself with the nature and order of the decisions considered at each step as well as the fact that they are small. See [22] for an analysis of two alternative decompositions of a KWIC system similar to the one defined here.

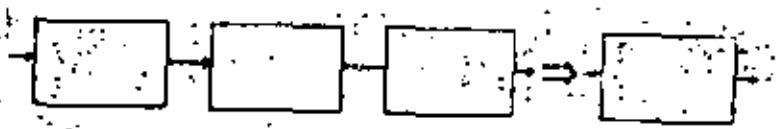
ants of initialization, test before or after the loop body, etc. These variants would not change the arguments to follow and have been omitted.)



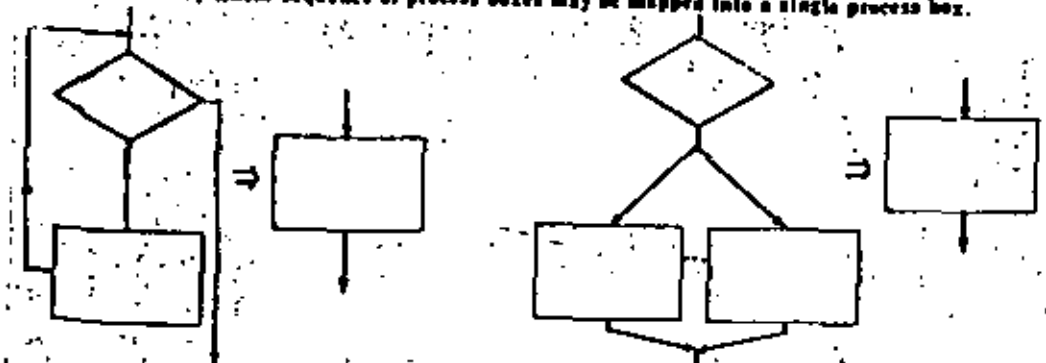
simple loop

case

Consider the following three transformations (T1, T2, T3) defined on arbitrary flow charts:



T1. Any linear sequence of process boxes may be mapped into a single process box.



T2. Any simple loop may be mapped into a process box.

T3. Any n-way "case" construct may be mapped into a process box.

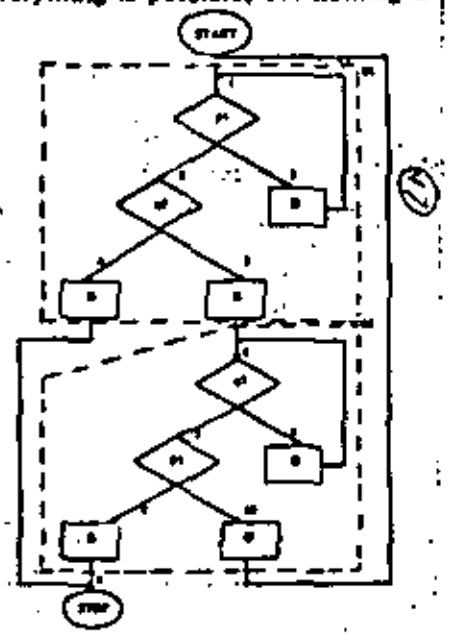
Any graph (flow chart) which may be derived by a sequence of these transformations we shall call a "reduced" form of the original. We shall say that a graph which may be reduced to a single node by some sequence of transformations is "goto-less" (independent of whether actual goto statements are used in its encoding) and that the sequence of transformations defines a set of nested "control environments." The sequence of transformations applied in order to reduce a graph to a single node may be used as a guide to both understanding and proving the correctness of the program [2, 4, 6, 7, 19].

The property of being "goto-less" in the sense defined above is a necessary condition for the program to have been designed by the constructive methodology. Moreover, the property depends only upon the topology of the program and not on the primitives from which it is synthesized; in particular, a goto statement might have been used. However, not only can such programs be constructed without a goto if conditionals and loops are available, but *any* use of a goto which is not equivalent to one of these will destroy the requisite topology. Hence any set of restrictions (on the use of the goto) which is intended to achieve this topology is equivalent to eliminating the goto.

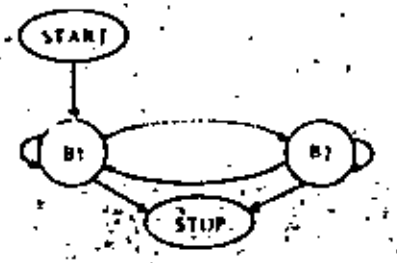
The theoretical possibility of eliminating the GOTO

It is possible to express the evaluation of an arbitrary computable function in a notation which does not have an explicit goto. This is not particularly surprising since: (1) several formal systems of computability theory, e.g., recursive functions, do not use the concept; (2) (pure) LISP does not use it; and (3) van Wijngaarden [13], in defining the semantics of Algol, eliminated labels and goto's by systematic substitution of procedures. However, this does not say that an algorithm for the evaluation of these functions is especially convenient or transparent in goto-less form. Alan Perlis has referred to similar situations as the "Turing Tarpit" in which everything is possible, but nothing is easy.

Knuth and Floyd [14] and Ashcroft and Manna [15] have shown that given an arbitrary flow chart it is *not* possible to construct another flow chart (using the same primitives and no additional variables) which performs the same algorithm and uses only simple conditional and loop constructs; of course other algorithms exist that compute the same function and which can be expressed with only simple conditionals and loops. The example given in Ashcroft and Manna of an algorithm which cannot be written in goto-less form without adding additional variables is:



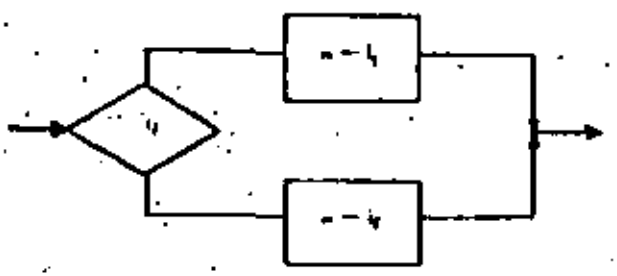
By enclosing some of the regions of the flow chart in dotted lines and labeling them (B1 and B2) as shown on the previous page, and further abstracting from the details of the process and decision structure, the abstract structure of this example is:



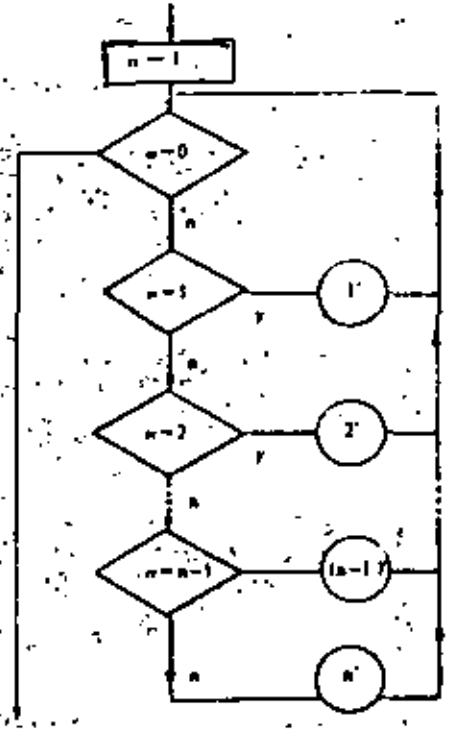
The reader is referred to [15] for a proof that such programs cannot be constructed from simple looping and conditional constructs unless an additional variable is added. Intuitively, however, it should be clear from the abstraction of the example that neither B1 nor B2 is inherently nested within the other. Moreover, the existence of multiple exit paths from B1 and B2 make it impossible to impose a superior (simple) loop (which inherently has a single exit path) to control the iteration between them unless some mechanism for path selection (e.g., an additional variable) is introduced.

In [21] Böhm and Jacopini show that an arbitrary flow chart program may be translated into an equivalent one with a single "while statement" by introducing new boolean variables, predicates to test them, and assignment statements to set them. A variant of this scheme involving the addition of a single integer variable, call it "α," which serves as a "program counter," is given below.

Suppose some flow chart program contains a set of process boxes assigned arbitrary integer labels $i_1, i_2, \ldots, i_n$, and decision boxes assigned arbitrary integer labels $i_{n+1}, i_{n+2}, \ldots, i_m$. (By convention assume the STOP box is assigned the label zero, and the entry box is assigned the label one.) For each process box, i_j, create a new box, i_j', identical to the former except for the addition of the assignment "$\alpha \leftarrow i_k$" where i_k is the label of the successor of i_j in the original program. For each decision box, i_p, create the macro box, i_p':

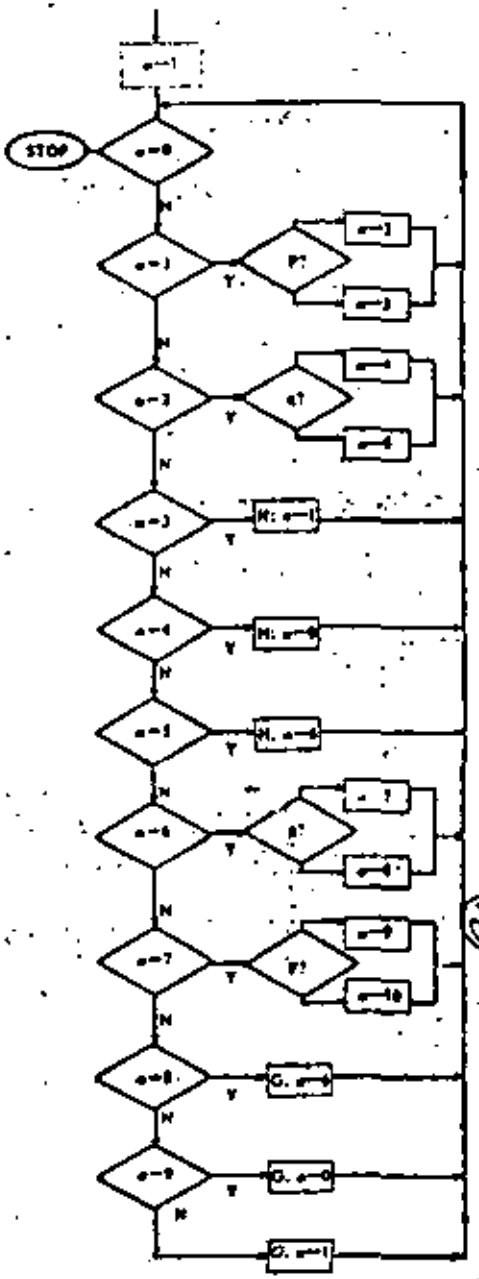


where i_t and i_f are the labels of the successors of the true and false branches of the decision box, i_p, in the original program. Now create the following flow chart:



Thus, for example, the Ashcroft and Manna example given earlier (the labels are given on the earlier diagram) becomes (see right):

Constructions such as the one given at right are undesirable not only because of their inefficiency, but because they destroy the topology (loop structure) and locality of the original program and thus make it extremely difficult to understand. Nevertheless, the construction serves to illustrate the point that adding (at least one) control variable is an effective device for eliminating the goto. Ashcroft and Manna have given algorithms for translating arbitrary programs into goto-less form (with additional variables) which preserve the efficiency and topology of the original program.



The practical possibility of eliminating the GOTO

As discussed in the previous section, it is theoretically possible to eliminate the goto. Moreover, there can be little quarrel with the *objectives* of the constructive programming methodology. A consequence of the particular methodology presented above is that it produces goto-less programs, thus the goto is unnecessary in programs produced according to this methodology. A key, perhaps the key, issue, then, is whether it is *practical* to remove the goto. In particular there is an appropriate suspicion among practicing programmers* that coding without the goto is both inconvenient and inefficient. In this section we shall investigate these two issues, for, if it is inconvenient or grossly inefficient to program without the goto then the practicality of the methodology is in question.

Convenience:

Programming without the goto is *not* (necessarily) inconvenient. The author is one of the designers, implementors, and users of a "systems implementation language," Bliss [16, 17, 18]; Bliss does not have goto. The language has been in active use for three years; we have thus gained considerable practical experience programming without the goto. This experience spans many people and includes several compilers, a conversational programming system (APL), an operating system, as well as numerous applications programs.

The inescapable conclusion from the Bliss experience is that the purported inconvenience of programming without a goto is a myth! Programmers familiar with languages in which the goto is present go through a rather brief and painless adaptation period. Once past this adaptation period they find that the lack of a goto is not a handicap; on the contrary, the invariant reaction is that the enforced discipline of programming without a goto structures and simplifies the task.

Bliss is not, however, a simple goto-less language; that is, it contains more than simple while-do and if-then-else (or case) constructs. There are natural forms of control flow that occur in real programs which, if not explicitly provided for in the language, either require a goto so that the programmer may synthesize them, or else will cause the programmer to contort himself to mold them into a goto-less form (e.g., in terms of the construction in the previous section). Contortion obscures and is therefore antipathetic with the constructive philosophy; hence the approach in Bliss has been to provide explicit forms of these natural constructs which are also inherently well-structured. In [19] the author analyzes the forms of control flow which are not easily realized in a simple goto-less language and uses this analysis to motivate the facilities in Bliss. Here we shall merely list some of the results of that analysis as they manifest themselves in Bliss (and might manifest themselves in any goto-less language):

1. A collection of "conventional" control structures: Many of the inconveniences of a simple goto-less language are eliminated by simply providing a fairly large collection of more-or-less "conventional" control structures. In particular, for example, Bliss includes: control "scopes" (blocks and compounds), conditionals (both if-then-else and case forms), several looping constructs (including while-do, do-while, and stepping forms), potentially recursive procedures, and co-routines.

2. Expression Language: As noted in an earlier section, one mechanism for expressing algorithms in goto-less form is through the introduction of at least one additional variable. The value of this variable serves to encode the state of the computation and direct subsequent flow. This is a common programming practice used even in languages in which the goto is present (e.g., the FORTRAN "computed goto"). Bliss is an "expression language" in the sense that every construct, including those which manifest control, is an expression and computes a value. The value of an expression (e.g., a block or loop) forms a natural and convenient implicit state variable.

3. Escape Mechanism: Analysis of real programs strongly suggests that one of the most common "good" uses of a goto is to prematurely terminate execution of a control environment — for example, to exit from the middle of a loop before the usual termination condition is satisfied. To accommodate this form of control Bliss* allows any expression (control environment) to be labeled; an expression of the form "leave <label> with <expression>" may be executed within the scope of this labeled environment. When a leave expression is executed two things happen: (1) control immediately passes to the end of the control environment (expression) named in the leave, and (2) the value of the named environment is set to that of the <expression> following the with. Note that the leave expression is a restricted form of forward branch just as the various forms of loop constructs are restricted backward jumps. In both cases the constructs are less general, and less dangerous, than the general goto.

*Including this author when he first read Dijkstra's letter in 1968.

*A somewhat different form of the Bliss escape is described in [19]; the form described in [17] has been replaced by that described above.

In summary, then, our experience with Bliss supports the notion that programming without the goto is no less convenient than with it. This conclusion rests heavily on the assumption that the goto was not merely removed from some existing language, but that a coherent selection of well-structured constructs were assembled as the basis of the control component of the new language. It would be unreasonable to expect that merely removing the goto from an existing language, say FORTRAN or PL/I, would result in a convenient notation. On the other hand, it is *not* unreasonable to expect that a relatively small set of additions to an existing language, especially the better structured ones such as Algol or PL/I, could reintroduce the requisite convenience. While not a unique set of solutions, the control mechanisms in Bliss are one model on which such a set of additions might be based.

Efficiency

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity. One of these sins is the construction of a "rat's nest" of control flow which exploits a few common instruction sequences. This is precisely the form of programming which must be eliminated if we are ever to build correct, understandable, and modifiable systems.

There *are* applications (e.g., "real time" processing) and there *are* (a few) portions of every program where efficiency is crucial. This is a real issue. However, the appropriate mechanism for achieving this efficiency is a highly optimizing compiler, not incomprehensible source code. In this context it is worth noting another benefit of removing the goto — a benefit which the author did not fully appreciate until the Bliss compiler was designed — namely, that of global optimization. The presence of goto in a block-structured language with dynamic storage allocation forces runtime overhead for jumping out of blocks and procedures and may imply a distributed overhead to support the possibility of such jumps. Eliminating the goto removes both of these forms of overhead. More important, however, is that: (1) the scope of a control environment is statically defined, and (2) all control appears as one of small set of explicit control constructs. A consequence of (1) is that the Fortran-H compiler [20], for example, expends a considerable amount of effort in order to achieve roughly the same picture of overall control as that implicit in the text of a Bliss program. The consequence of (2) is that the compiler need only deal with a small number of well-defined control forms; thus failure to optimize a peculiarly constructed variant of a common control structure is impossible. Since flow analysis is pre-requisite to global optimization, this benefit of eliminating the goto must not be underestimated.

Summary

One goal of our profession must be to produce large programs of predictable reliability. To do this requires a methodology of program construction. Whatever the precise shape of this methodology, whether the one sketched earlier or not, one property of that methodology must be to isolate (sub) components of a program in such a way that the proof of the correctness of an abstraction from those components can be made independent of both their implementation and the context in which they occur. In particular this implies that unrestricted branching between components cannot be allowed.

Whether or not a language contains a goto and whether or not a programmer uses a goto in some context is related, in part, to the variety and extent of the other control features of the language. If the language fails to provide important control constructs, then the goto is a crutch from which the programmer may synthesize them. The danger in the goto is that the programmer may do this in obscure ways. The advantage in eliminating the goto is that these same control structures will appear in regular and well-defined ways. In the latter case, both the human and the compiler will do a better job of interpreting them.

References

1. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.

2. _____, "A Constructive Approach to the Problem of Program Correctness," *BIT*, Vol. 8, No. 3 (1968), pp. 174-86.

3. _____, "Structured Programming," *Software Engineering, Concepts and Techniques: Proceedings of the NATO Conferences*, eds: P. Naur, B. Randell, and J.N. Buxton (New York: Petrocelli/Charter, 1976), pp. 222-26.

4. _____, *Notes on Structured Programming*, 2nd ed., Technische Hogeschool Eindhoven, Report No. EWD-248, 70-WSK-0349 (Eindhoven, The Netherlands: April 1970); see also *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).

5. P. Naur, "Proof of Algorithms by General Snapshots," *BIT*, Vol. 6, No. 4 (1966), pp. 310-16.

6. _____, "Programming by Action Clusters," *BIT*, Vol. 9, No. 3 (1969), pp. 250-58.

7. C.A.R. Hoare, "Proof of a Program: FIND," *Communications of the ACM*, Vol. 14, No. 1 (January 1971), pp. 39-45.

8. N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No. 4 (April 1971), 221-27.

9. D.L. Parnas, "Information Distribution Aspects of Design Methodology," *Proceedings of the 1971 IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1971), pp. 339-44; see also Carnegie-Mellon University, Computer Science Technical Report (Pittsburgh: 1971).

10. J. King, "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, 1969.

11. Z. Manna, "Termination of Algorithms," Ph.D. Thesis, Carnegie-Mellon University, 1968.

12. _____, "The Correctness of Programs," *Journal of Computer & System Sciences*, Vol. 3 (May 1969), pp. 119-27.

13. A. van Wijngaarden, "Recursive Definition of Syntax and Semantics," *Formal Language Description Languages*, ed. T.B. Steel. (Amsterdam, The Netherlands: North-Holland Publishing Co., 1966).

14. D. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31, 177; see also Stanford University, Computer Science Technical Report, Vol. CS148 (Stanford, Calif.: January 1970).

15. E. Ashcroft and Z. Manna, "The Translation of 'go to' Programs to 'while' Programs," *Proceedings of the 1971 IFIP Congress*, Vol. I (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 250-55; see also Stanford University, AI Memo AIM-138, STAN CS-71-88 (Stanford, Calif.: January 1971).

16. W.A. Wulf, et al., *BLISS11 Reference Manual*, Carnegie-Mellon University, Computer Science Department Report No. AD-739964 (Pittsburgh: March 1972).

17. W.A. Wulf, D.B. Russell, and A.N. Habermann, "BLISS: A Language for Systems Programming," *Communications of the ACM*, Vol. 14, No. 12 (December 1971), pp. 780-90.

18. W.A. Wulf, et al., "Reflections on a Systems Programming Language," *Proceedings of the SIGPLAN Symposium on Systems Implementation Languages* (October 1971).

19. W.A. Wulf, "Programming without the GOTO," *Proceedings of the 1971 IFIP Congress*, Vol. 1 (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 408-13.

20. E.S. Lowry and C.W. Medlock, "Object Code Optimization," *Communications of the ACM*, Vol. 12, No. 1 (January 1969), pp. 13-22.

21. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.

22. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12 (December 1971), pp. 1053-58.

23. Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," *Communications of the ACM*, Vol. 16, No. 8 (August 1973), pp. 491-502.

24. R. Burstall, "An Algebraic Description of Programs with Assertions, Verification and Simulation," *Proceedings of the ACM Conference on Proving Assertions About Programs, SIGPLAN Notices*, Vol. 7, No. 1 (January 1972), pp. 7-14.

A Case for the GOTO

Introduction

It is with some trepidation that I undertake to defend the goto statement, a construct which while ancient and much used has been shown to be theoretically unnecessary [1] and in recent years has come under so much attack [2]. In my opinion, there have been far too many goto statements in most programs, but to say this is not to say that goto should be eliminated from our programming languages. This paper contains a plea for the retention of goto in both current and future languages. Let us first examine the context in which the controversy occurs.

A wise philosopher once pointed out to a lazy king that there is no royal road to geometry. After discovering, in the late fifties, that programming was *the* computer problem, a search was made during the sixties for the royal road to programming. Various paths were tried including comprehensive operating systems, higher level languages, project management techniques, time sharing, virtual memory, programmer education, and applications packages. While each of these is useful, they have not solved the programming problem. Confronted with this unresolved problem and with few good ideas on the horizon, some people are now hoping that the royal road will be found through style, and that banishment of the goto statement will solve all. The existence of this controversy and the seriousness assigned to it by otherwise very sensible people are symptoms of a malaise in the computing community. We have few promising new ideas at hand. I also suspect that the controversy reflects something rather deep in human nature, the notion that language is magic and the mere utterance of certain words is dangerous or defiling. Is it an accident that "goto" has four letters?

Having indicated my belief that this controversy is not quite as momentous as some have made out, it is appropriate to point out some beneficial aspects. First, interest has been focused on programming style and while style is not everything it does have a great deal of importance. Second, the popularity of the no goto rule is, in large part, due to the fact that it is a simple rule which does improve the code produced by most programmers. As we shall see, this is not sufficient grounds for banishing the construct from our languages, although it may well justify teaching alternative methods of programming to beginners or restricting its use on a project. Perhaps the most beneficial aspect of the controversy will be to encourage the use of block structure languages and to discourage use of our most popular languages, COBOL and FORTRAN as they are not well suited to programming without the goto.

The principal motivation behind eliminating the goto statement is the hope that the resulting programs will not look like a bowl of spaghetti. Instead they will be simple, elegant and easy to read, both for the programmer who is engaged in composition and checkout as well as the poor soul attempting future modification. By avoiding goto one guarantees that a program will consist entirely of one-in-one-out control structures. It is easy to read a properly indented program without goto statements, which is written in a block structure language. The possible predecessors of every statement are obvious and, with the exception of loops, all predecessors are higher on the page. (I assume nobody writes inner procedures longer than a page anymore?) Why then should we retain the goto statement in our current and future programming languages?

Theoretical considerations

It has been demonstrated that there are flow charts which cannot be converted to a procedural notation without goto statements [3, 4]. It turns out though that this result is not really an argument for the retention of goto as there are means by which a procedure can be *rewritten* in a systematic manner to eliminate all instances of goto. An almost trivial method is to introduce a new variable which can be thought of as the instruction counter along with tests and sets of this counter. The method is fully described by Böhm and Jacopini [1]. The results of this procedure when applied to a large program with many instances of goto would usually be a program which is less readable than the original program with goto statements. However, nobody seems to be advocating using such unconsidered methods.

The real issue is that theoretical work has suggested a number of techniques that can be used to rewrite programs, eliminating the instances of goto. These include replication of code (node splitting), the introduction of new variables and tests as well as the introduction of procedures. Any of these techniques, when used with discretion, can increase the readability of code. The question is whether there are any instances when the application of such methods decreases clarity or produces some other undesirable effect. Whether or not to retain the goto does not seem to be a theoretical issue. It is rather a matter of taste, style and the practical considerations of day to day computer use.

Alternatives to GOTO

With respect to current languages which are in wide use such as COBOL and FORTRAN, there is the practical consideration that the goto statement is necessary. Even where a language is reasonably well-suited to programming without the goto, the elimination of this construct may be at once too loose and too restrictive. PL/I provides some interesting examples here. One exits from an Algol procedure when the flow of control reaches the end bracket. PL/I provides an additional mechanism, an explicit RETURN statement. Consider the table look up in Fig. 1 which is similar to an example of Floyd and Knuth [4]. The problem is to find an instance of X in the vector A and if there is no instance of X in A, then make a new entry in A of the argument X. In either case the index of the entry is returned. A count of the number of matches associated with each entry in A is also maintained in an associated vector, B. In this example there are no goto statements but the two RETURN statements cause an exit from both the procedure and the iterative DO. Thus the procedure has control structures which have more than one exit and one-in-one-out control structures were a principal reason for avoiding goto. Should the PL/I programmer add a rule forbidding RETURN? The procedure could then be rewritten as in Fig. 2. This involves the introduction of a new variable, SWITCH, and a new test. If one assumes that the introduction of gratuitous identifiers and tests is undesirable perhaps RETURN is a desirable construct even though it can result in multiple exit control structures. It is my feeling that procedures with several RETURN statements are easy to read and modify because they follow the top to bottom pattern and maintain the obvious predecessor characteristic, while avoiding the introduction of new variables. RETURN is therefore preferable to the alternative of introducing new variables and tests.

However, RETURN is a very specialized statement. It only permits an exit from one level of one type of control, the procedure. One could generalize the construct to apply to multiple levels of control and to DO groups or BEGIN blocks as well as procedures. This is exactly the flavor of the BLISS leave [6] construct. Lacking such language, the PL/I user must content himself with goto. But is this a bad thing? The good programmer, who understands the potential complexity which results from excessive use of goto, will attempt to resist such an algorithm. Failing to find an elegant restatement, he will insert the label and its associated goto out of the desired control structure. The label stands there as a warning to the reader of the routine that this is a procedure with more than the usual complexity. Note also that the label point catches the eye. It is immediately apparent when looking at this statement that it has an unusual predecessor. The careful reader will want to consult a cross reference listing to determine the potential flow of control. Note that the BLISS leave construct is somewhat less than ideal here. In BLISS when one examines the code which follows a bracket terminating a level of control, its potential predecessors are not immediately apparent. One must look upward on the page for its associated label, which indicates a potential unusual predecessor and then find the leave. It is my feeling that unusual exits from levels of control should be avoided. The multiple level case is especially ugly. Where such constructs are necessary, it should be made completely obvious to all. Statements such as the BLISS leave encourage unusual exits from multiple levels of control. One should not cover up the fact that there is an awkward bit of logic by the introduction of a new control construct.

```
LOOK_UP:
    PROC (X);
        DO I = 1 TO A_TOP;
            IF A(I) = X THEN
                DO;
                    B(I) = B(I) + 1;
                    RETURN (I);
                END;
        END;
        A(I) = X;
        B(I) = 1;
        A_TOP = A_TOP + 1;
        RETURN (I);
    END;
```

Figure 1

```
LOOK_UP2:
    PROC (X);
        SWITCH = 1;
        DO I = 1 TO A_TOP WHILE (SWITCH = 1);
            IF A(I) = X THEN
                SWITCH = 0;
        END;
    IF SWITCH = 0 THEN
        B(I) = B(I) + 1;
    ELSE
        DO;
            A(I) = X;
            B(I) = 1;
            A_TOP = A_TOP + 1;
        END;
    RETURN(I);
END;
```

Figure 2

Another interesting PL/I control construct is the ON unit. This is a named block which is automatically invoked on certain events such as overflow, but it can also be invoked explicitly by a statement of the form:

SIGNAL CONDITION (name);

The name is established dynamically and need not be declared in the scope of the SIGNAL. This facility often eliminates the need to pass special error return parameters or test a return code which indicates abnormal termination of a lower level procedure. After completion of an ON unit activated by SIGNAL, control is passed back to the statement following the SIGNAL. This is usually not useful. One wants to terminate the signaling block and the only way to do this in PL/I is with a goto out of the ON unit. Is SIGNAL permissible under the no goto criteria? Elimination of goto seems too restrictive here as SIGNAL is a useful facility which can eliminate much messy programming detail. However, the natural consequence of using the SIGNAL statement is to terminate an ON unit with a goto. Perhaps it is best to admit that there is no very good alternative to a goto statement in this situation.

GOTO as a basic building block

The lack of a case statement in PL/I is a clear deficiency. The resourceful programmer will construct one out of a goto. This does not make up for the lack of a case statement, but it does point up an interesting and highly legitimate use of goto. One can use it as a primitive to construct more advanced and elegant control structures. Imaginative programmers will, from time to time, develop new control constructs as Hoare invented the case statement [5]. Those that are worthwhile will be informally defined and implemented with a macro preprocessor. The better ones will appear in experimental compilers and eventually the best will find their way into the standard languages. Such inventions are often very hard to implement with macro preprocessors for existing languages without use of the goto construct. There is still room for the incorporation of unusual control mechanisms into existing block structure languages. Decision tables are a prime example. One way of handling decision tables is to have a preprocessor convert them to source language statements. If a convenient translation process introduces goto statements, this is not important as the basic documentation is at the decision table level. The source language is treated as an internal language. The ease of translation is more important than the introduction of goto statements.

Another related reason for retaining goto even in our newest languages is that it is often possible to use a language as the target to which one translates a secondary source language. If the secondary language has goto or even a different set of control constructs, then translation could be very difficult without a goto in the target language. In other words source languages and their associated compilers are useful building blocks for the development of special constructs or languages and elimination of goto decreases the range of usefulness of a language.

GOTO as an escape

Part of the reason for retaining goto is that the world is not always a very elegant place and sometimes a goto is a useful, if ugly, tool to handle an awkward situation. Algorithms are often messy. Sometimes this may be due to inherent complexity. I suspect, however, that most of the time it is because not enough time or intelligence has been applied. Where time or intelligence are lacking, a goto may do the job. Every program will not be published. Many may be used only once. I tend to sympathize with the programmer who fixes up a one time program at 3:00 a.m. with a goto. Of course, there is always the danger that the programmer will lapse into bad habits but I am willing to take that chance. Perhaps it is an opportunity, for when the intelligent supervisor reads the code of those under him, he can focus on any goto statements. A programmer should be able to justify each use of goto.

I have avoided discussing performance, which like death and taxes, none of us can avoid forever. Suppose a procedure runs too slowly or takes up too much space. A rewrite of the procedure or restructuring of the data may be in order. But if that fails one may be driven to a rewrite in assembly language. There is an intermediate alternative which may solve the problem without resort to an assembler. The programmer who writes structured programs uses certain techniques such as the introduction of procedures and the repetition of code which can result in the loss of time and space. Given the idiosyncracies of many compilers, a little reorganization of code and a few goto statements inserted by a clever programmer can often improve performance. This is not a practice which I recommend for those starting a project, even where it is known to have stringent performance requirements. One should give up a structured program in a higher level language only after performance bottlenecks have been clearly identified and then only give up what is absolutely necessary. My guess is that very few such situations will exist but when they do, a slightly contorted procedure in a higher level language may be an attractive alternative to one written in assembly language. The villain here is the compiler which produces bad code in some situations. Would elimination (as opposed to avoidance) of goto significantly ease the task of compiler writers and thus help us to get better object code? It is difficult to do justice to this problem as there are so many different compiling techniques and some would be helped and some would not. My feeling is that elimination of the goto would not dramatically ease the problems of compiler writers. Even in compilers which do extensive control flow analysis, a small percentage of implementation effort is devoted to that task. A more interesting subject for compiler writers is the identification of those optimizations which improve the performance of programs written with none or very few goto statements. Viewed in this light the

existence of well-structured programs imposes an additional obligation and more work on compiler writers. This is work which they should eagerly accept so that programmers will not have to make the trade-off between a well-structured program and one that performs well. More work is required in this area.

Varieties of programming style

The goto issue is part of the larger topic of overall programming style. One of my worries is that we will become the prisoners of one currently fashionable "classical" style. Perhaps other rules of style are better. For example we might say that only a goto which was directed forward was elegant. Perhaps it is useful to restrict ourselves to standard type labels such as "PROCEXIT". Vagaries of style or fashion need not disturb students who should be taught in a rather constrained way which is established by the teacher. Also, those working on large projects will have to conform to standards. However, experienced programmers and language designers of taste and imagination will want to experiment and they should be encouraged to do so. APL provides an interesting example of a diverse style. A computed goto, in the form of a right pointing arrow exists in APL, but other than function invocation there are no control constructs such as IF THEN ELSE or an iteration statement. Surprisingly one does not get a maze of goto statements in a well-written APL function, for the powerful array operators can be used in situations where loops occur in other languages. Sequential execution of statements thus becomes the general rule and few right pointing arrows are required. Whether an algorithm written in APL is clearer than the same algorithm written in a block structure language seems to be a matter on which intelligent people of taste will disagree.

Elegance in programming involves more than avoiding goto, and beyond the goto controversy there are a great many other important issues of style. There is the question about the clarity of array operations in APL and PL/I, as well as structure operations in COBOL and PL/I. To what extent are implicit conversions a subsumption of extraneous detail and in what instances do they produce surprising results? There are many questions about optimal size and complexity with respect to expressions, nesting of IF and iteration statements as well as the size and complexity of procedures. To what extent do declarations properly subsume detail and to what extent do they leave the meaning of a statement unclear unless one is simultaneously examining the declaration? Under what circumstances, if any, should functions have side effects or should iteration replace recursion? To what extent can we eliminate assignment? These and other questions are subtle but important stylistic problems which we are likely to pass over if we concentrate too heavily on the relatively simple and unimportant issue of goto.

References

1. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71.
2. E.W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-48.
3. E. Ashcroft, and Z. Manna, "The Translation of 'go to' Programs to 'while' Programs," *Proceedings of the 1971 IFIP Congress*, Vol. I (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 250-55; see also Stanford University, AI Memo, AIM-138, STAN CS-71-88 (Stanford, Calif.: January 1971).
4. D.E. Knuth and R.W. Floyd, "Notes on Avoiding 'go to' Statements," *Information Processing Letters*, Vol. 1, No. 1 (February 1971), pp. 23-31; see also Stanford University, Computer Science Technical Report, Vol. CS148, (Stanford, Calif.: January 1970).
5. N. Wirth and C.A.R. Hoare, "A Contribution to the Development of ALGOL," *Communications of the ACM*, Vol. 9, No. 6 (June 1966), pp. 413-32.
6. W.A. Wulf, D.B. Russell, and A.N. Habermann, "BLISS: A Language for Systems Programming," *Communications of the ACM*, Vol. 14, No. 12 (December 1971), pp. 780-90.

Conclusion

goto should be retained in both current and future languages because it is useful in a limited number of situations. Programmers should work hard to produce well-structured programs with one-in-one-out control structures which have no goto statements. Where this is not possible, we should not think that elegance is achieved with a magic language formula. It is far better to admit the awkwardness and use the goto. Furthermore, goto is a useful means to synthesize more complex control structures and increases the usefulness of a language as a target to which other languages can be translated. Viewed in the light of practical programming as an ultimate escape, goto can also be justified if not encouraged. Finally our wisdom has not yet reached the point where future languages should eliminate the goto. If future work indicates that by avoiding goto we can gain some important advantage such as routine proofs that programs are correct, then the decision to retain the goto construct should be reconsidered. But until then, it is wise to retain it.

On the Composition of Well-Structured Programs

Introduction

In the first decade of computers, say up to the early sixties, computers were quite limited in their power. The task of the programmer was to formulate algorithms in the specific order codes of these machines so that they were utilized as effectively as possible. Primarily because of their limitations, this task was achieved by collecting sets of clever techniques and startling tricks, and by finding applications for them as frequently as possible. Examples of such techniques were the programmed self-modification of parts of the program, such as, for instance, the conversion of conditional jumps into dummy instructions and vice versa, or the sharing of store for functionally independent, but never simultaneously used auxiliary variables.

Tricks were necessary at this time, simply because machines were built with limitations imposed by a technology in its early development stage, and because even problems that would be termed "simple" nowadays could not be handled in a straightforward way. It was the programmers' very task to push computers to their limits by whatever means available. We should recall that the absence of index registers (and indirect addressing), for example, made automatic code modification a mere necessity (see also [1, 2]).

The essence of programming was understood to be the *optimization of the efficiency* of particular machines executing particular algorithms. As computers grew more powerful, the problems posed to the programmers grew proportionally, and as a result, the growing power of hardware did not ease, but rather increased the burden. The elimination of deficiencies, errors and blunders — called debugging — became the overwhelming problem.

Understandably, the remedy was sought in the development and use of better tools in the form of programming languages. The amount of resistance and prejudices which the farsighted originators of FORTRAN had to overcome to gain acceptance of their product is a memorable indication of the degree to which programmers were preoccupied with efficiency, and to which trickology had already become an addiction. However, once these adversities and fears had been overcome, FORTRAN had a tremendous impact — an impact that is still felt today. ALGOL 60 followed several years later; it went beyond FORTRAN in several significant respects, but essentially shared the same purpose and intention. In particular, it extended to the level of statements what FORTRAN had introduced on the level of (arithmetic) expressions: *structure*. But ALGOL 60 was not very successful when measured by its frequency of use in technical and commercial applications. There are many reasons for this, one being that it appeared on the scene when the relevance of structure had not yet been widely recognized, and its restrictiveness against the use of clever tricks was considered to be a handicap and a deficiency. The law of the "Wild West of Programming" was still held in too high esteem! The same inertia that kept many assembly code programmers from advancing to use FORTRAN is now the principal obstacle against moving from a "FORTRAN style" to a structured style.

As the power of computers on the one side, and the complexity and size of the programmer's task on the other continued to grow with a speed unmatched by any other technological venture, it was gradually recognized that the true challenge does not consist in pushing computers to their limits by saving bits and microseconds, but in being capable of organizing large and complex programs, and assuring that they specify a process that for all admitted inputs produces the desired results. In short, it became clear that any amount of efficiency is worthless if we cannot provide *reliability* [4]. But how can this reliability be provided? Here structure enters the scene as the one essential tool for mastering complexity, the effective means of converting a seemingly senseless mass of bits or characters into meaningful and intelligible information. We must recognize the strong and undeniable influence that our language exerts on our ways of thinking, and in fact defines and delimits the abstract space in which we can formulate — give form to — our thoughts.

But now the term *structured programming* has been coined, and it seems finally to be achieving what the term "structured language" was unable to suggest. It was first used by E.W. Dijkstra [3], and has spread with various interpretations and connotations since then. It is the expression of a conviction

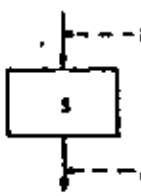
that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically, and that particular techniques should be replaced (or augmented) by a method. At its heart lies an *attitude* rather than a recipe: the admission of the limitations of our minds. The recognition of these limitations can be used to our advantage, if we carefully restrict ourselves to writing programs which we can manage intellectually, where we fully understand the totality of their implications.

1. Intellectual manageability of programs

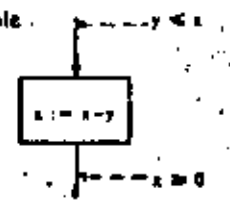
Our most important mental tool for coping with complexity is *abstraction*. Therefore, a complex problem should not be regarded immediately in terms of computer instructions, bits, and "logical words," but rather in terms and entities natural to the problem itself, abstracted in some suitable sense. In this process, an abstract program emerges, performing specific operations on abstract data, and formulated in some suitable notation — quite possibly natural language. The operations are then considered as the constituents of the program which are further subjected to decomposition to the next "lower" level of abstraction. This process of *refinement* continues until a level is reached that can be understood by a computer, be it a high-level programming language, FORTRAN, or some machine code [5, 6].

For the intellectual manageability, it is crucial that the constituent operations at each level of abstraction are connected according to sufficiently simple, well understood *program schemas*, and that each operation is described as a piece of program with *one starting point* and a *single terminating point*. This allows defining states of the computation (P, Q), i.e., relations among the involved variables, and attaching them to the starting and terminating points of each operation (S). It is immaterial, at this point, whether these states are defined by rigorous mathematical formulas (i.e., by predicates of logical calculus) or by sufficiently clear and informative sentences, or by a combination of both. The important point is that the programmer has the means to gain clarity about the interface conditions between the individual building blocks out of which he composes his program [7].

(1)



Example



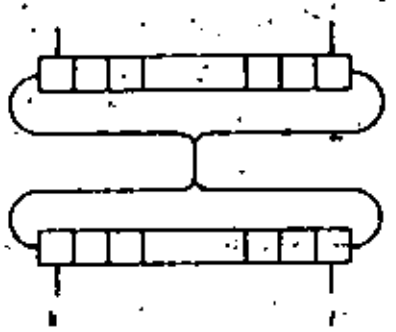
An example may clarify the issues at this point. The reader should be aware that any example that is sufficiently short to fit onto a single page cannot be much more than a metaphor, probably unconvincing to habitual skeptics. The important thing is to abstract from the example and to imagine the same method being applied to large programming problems.

Example 1: Sequential merging

Given a set of $n = 2^N$ integer variables $a_1 \cdots a_n$, find a recipe to permute their values such that $a_1 \leq a_2 \leq \cdots \leq a_n$ using the principle of sequential merging. Thus, we are to sort under the assumption of strictly sequential access. Briefly told, we shall use the following algorithm:

1) Pick individual components $a_i^{(1)}$ and merge them into ordered pairs, depositing them in a variable $a^{(2)}$.
2) Pick the ordered pairs from $a^{(2)}$ and merge them pairwise into ordered quadruples, depositing them in a variable $a^{(3)}$.
3) Continue this game, each time doubling the size of the merged subsequences, until a single sequence of length $n = 2^N$ is generated.

(2)



At the outset, we notice that two variables $a^{(1)}$ and $a^{(2)}$ suffice, if the items are alternately shuttled between them. We shall introduce a single array variable A with $2n$ components, such that $a^{(1)}$ is represented by $A[1] \cdots A[n]$ and $a^{(2)}$ is represented by $A[n+1] \cdots A[2n]$. Each of these two conceptually independent parts has two points of sequential access, or read/write heads. These are to be denoted by pairs of index variables i, j and k, l respectively. We may now visualize the sort process as a repeated transfer under merging of tuples *up* and *down* the array A.

The first version of our program is evidently a repetition of the merge shuttle of p-tuples, where each time around p is doubled and the direction of the shuttle is changed. As a consequence, we need two variables, one to denote the tuple size, one to denote the direction. We will call them p and up. Note that each repetitive operation must contain a change of its (control) variables within the loop, an initialization in front of the loop, and a termination condition. We easily convince ourselves of the correctness of the following program:

```
up := true; p := 1;                                              (3)
repeat 1: "initialize indices i, j, k, and l";
       2: "merge p-tuples from i- and
           j-sequences into k- and
           l-sequences";
          up := ¬up; p := 2*p
until p = n
```

Statement-1 is easily expressed in terms of simple assignments depending on the direction of the merge pass:

```
1: if up then                                                    (4)
     begin i := 1; j := n;
           k := n+1; l := 2*n
     end
   else
     begin k := 1; l := n;
           i := n+1; j := 2*n
     end
```

Statement-2 describes the repeated merging of p-tuples; we shall control the repetition by counting the number m of items merged. The sources are designated by the indices i and j; the destination alternates between indices k and l. Instead of introducing a new variable standing alternately for k and l, we use the simple solution of interchanging k and l after each p-tuple merge, and letting k denote the destination index at all times. Clearly, the increment of k has then to alternate between the values $+1$ and -1; to denote the increment, we introduce the auxiliary variable h. We can easily convince ourselves that the following refinement is correct:

```
2: begin m := n; h := 1;                                         (5)
     repeat m := m-2*p;
       3: "merge one p-tuple from
           each of i and j to k, in-
           crement k after each
           move by h"; h := -h;
       4: "exchange k and l"
     until m = 0
   end
```

Whereas statement-4 is easily expressed as a sequence of simple assignments, statement-3 involves more careful planning. It describes the actual merge operation, i.e., the repeated comparison of the two incoming items, the selection of the lesser one, and the stepping up of the corresponding index. In order to keep track of the number of items taken from the two sources, we introduce the two counter variables q and r. It must be noted that the merge always exhausts only one of the two sources, and leaves the other one nonempty. Therefore, the leftover tail must subsequently be copied onto the output sequence. These deliberations quickly lead to the following description of statement-3:

```
3: begin q := p; r := p;                                         (6)
     repeat {select the smaller item}
       if A[i] ≤ A[j] then
         begin A[k] := A[i];
               k := k+h; i := i+1;
               q := q-1
         end
       else
         begin A[k] := A[j];
               k := k+h; j := j-1;
               r := r-1
         end
     until (q = 0) ∨ (r = 0);
     5: "copy tail of i-sequence";
     6: "copy tail of j-sequence"
   end
```

The manner in which the tail copying operations are stated demands that they be designed to have no effect, if initially their counter is zero. Use of a repetitive construct testing for termination *before* the first execution of the controlled statement is therefore mandatory.

```
5: while q ≠ 0 do                                                (7)
     begin A[k] := A[i];
           k := k+h;
           i := i+1; q := q-1
     end

6: while r ≠ 0 do
     begin A[k] := A[j];
           k := k+h;
           j := j-1; r := r-1
     end
```

This concludes the development and presentation of this program, if a computer is available to accept statements of this form, i.e., if a suitable compiler is available.

In passing, I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, "top-down" manner. Later refinement steps may often show that earlier decisions are inappropriate and must be reconsidered. But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness. The emerging modularity is particularly wel-

come if programs have to be adjusted to changed or extended specifications. This is a most essential advantage, since in practice few programs remain constant for a long time. The reader is urged to rediscover this advantage by generalizing this merge-sort program by allowing n to be any integer greater than 1.

Example 2: Squares and palindromes

List all integers between 1 and N whose squares have a decimal representation which is a palindrome. (A palindrome is a sequence of characters that reads the same from both ends.)

The problem consists in finding sequences of digits that satisfy two conditions: they must be palindromes, and they must represent squares. Consequently, there are two ways to proceed: either generate all palindromes (with $\log N^2$ digits) and select those which represent squares, or generate all squares and then select those whose representations are palindromes. We shall pursue the second method, because squares are simpler to generate (with conventional programming facilities), and because for a given N there are fewer squares than palindromes. The first program draft then consists of essentially a single repetitive statement.

```
n := 0;                                                    (8)
repeat n := n+1; generate square;
  if decimal representation of square
       is a palindrome
    then write n
until n = N
```

The next step is the decomposition of the complicated, verbally described statements into simpler parts. Obviously, before testing for the palindrome property, the decimal representation of the square must have been computed. As an interface between the individual parts we introduce auxiliary variables. They represent the result of one step and function as the argument of the successive step.

```
d[1] ... d[L]   an array of decimal digits
L               the number of digits computed
p               a Boolean variable
```

(note that $L = entier(2 \log N) + 1$)

The refined version of (8) becomes

```
n := 0;                                                    (9)
repeat n := n+1; s := n*n;
  d := decimal representation of s;
  p := d is a palindrome;
  if p then write (n)
until n = N
```

and we can proceed to specify the three component statements in even greater detail. The computation of a decimal representation is naturally formulated as the repeated computation of individual digits starting "at the right."

```
L := 0;                                                   (10)
repeat L := L+1;
  separate the rightmost digit of s,
    call it d[L]
until s = 0
```

The separation of the least significant digit is now easily expressed in terms of elementary arithmetic operations as shown in (12). Hence, the next task is the decomposition of the computation of the palindrome property p of d. It is plain that it also consists of the repeated, sequential comparison of corresponding digits. We start by picking the first and the last digits, and then proceed inwards. Let i and j be the indices of the compared digits.

```
i := 1; j := L;                                           (11)
repeat compare the digits;
  i := i+1; j := j-1
until (i ≥ j) or digits are unequal
```

A last refinement leads to a complete solution entirely expressed in terms of a conventional programming language with adequate structuring facilities.

```
n := 0;                                                   (12)
repeat n := n+1; s := n*n; L := 0;
  repeat L := L+1;
    r := s div 10; d[L] := s - 10*r;
    s := r
  until s = 0;
  i := 1; j := L;
  repeat p := d[i] = d[j];
    i := i+1; j := j-1
  until (i ≥ j) or ¬p;
  if p then write (n)
until n = N
```

This ends the presentation of Example 2.

2. Simplicity of composition schemes

In order to achieve intellectual manageability, the elementary composition schemes must be simple. We have encountered most of the truly fundamental ones in this second example. They encompass *sequencing*, *conditioning*, and *repetition* of constituent statements. I should like to elaborate on what is meant by simplicity of composition scheme. To this end, let us select as example the repetitive scheme expressed as

```
while B do S                                              (13)
```

It specifies the repeated execution of the constituent statement S, while — at the outset of each repetition — condition B is satisfied. The simplicity consists in the ease with which we can infer properties about the while statement from known properties of the constituent statement. In particular, assume that we know that S leaves a property P on its variables unchanged or *invariant* whenever B is true initially; this may be expressed formally as

$$P \wedge B \{S\} P \tag{14a}$$

according to the notation introduced by Hoare [8]. Then we may infer that the while statement also leaves P invariant, regardless of the number of times S was repeated. Since the repetition process terminates only after condition B has become false, we may infer that in addition to P, also $\neg B$ holds after the execution of the while statement. This inference may be expressed formally as

$$P \{\text{while } B \text{ do } S\} P \wedge \neg B \tag{14b}$$

This formula contains the essence of the entire while-construct. It teaches us to look for an invariant property P, and to consider the result of the repetition to be the logical combination of P and the negation of the continuation condition B. A similar pattern of inference governs the repeat-construct used in the preceding examples. Assuming that we can prove

$$Q \vee (P \wedge \neg B) \{S\} P \tag{15a}$$

about S, then we may conclude that

$$Q \{\text{repeat } S \text{ until } B\} P \wedge B \tag{15b}$$

holds for the repeat-construct.

There remains the question, whether all programs can be expressed in terms of hierarchical nestings of the few elementary composition schemes mentioned. Although in principle this is possible, the question is rather, whether they can be expressed conveniently, and whether they *should* be expressed in such a manner. The answer must necessarily be subjective, a matter of taste, but I tend to answer affirmatively. At least an attempt should be made to stick to elementary schemes before using more elaborate ones. Yet, the temptation to rescind this rule is real, and the chance to succumb is particularly great in languages offering a facility like the goto statement, which allows the instantaneous invention of any form of composition, and which is the key to any kind of structural irregularity.

The following short example illustrates a typical situation, and the issues involved.

Example 3: Selecting distinct numbers

Given is a sequence of (not necessarily different) numbers $r_0, r_1, r_2, \cdots$. Select the first n distinct numbers and assign them sequentially to an array variable a with n elements, skipping any number r_i that had already occurred. (The sequence r may, for instance, be obtained from a pseudo-random number generator; and we can rest assured that the sequence r contains at least n different numbers.)

An obvious formulation of a program performing this task is the following:

```
for i := 1 to n do                                              (16)
begin L: get(r);
  for j := 1 to i-1 do
    if a[j] = r then goto L;
  a[i] := r
end
```

It cannot be denied that this "obvious" solution has been suggested by the tradition of expressing a repeated action by a for statement (or a DO loop). The task of computing a value for a is decomposed into n identical steps of computing a single number $a[i]$ for $i = 1 \cdots n$. Another influence leading to this formulation is the tacit assumption that the probability of two elements of the sequence being equal is reasonably small. Hence, the case of a candidate r being equal to some $a[j]$ is considered as the exception: it leads to a break in the orderly course of operations and is expressed by a jump. The elimination of this break is the subject of our further deliberations.

Of course, the goto-statement may be easily — almost mechanically — replaced in a transcription process leading to the following goto-less version.

```
for i := 1 to n do                                              (17)
begin
  repeat get(r); ok := true;
      j := 1;
    while (j<i) ∧ ok do
      begin ok := a[j] ≠ r; j := j+1
      end
  until ok;
  a[i] := r
end
```

The transcription consists of the replacement of the for statement with a fixed termination condition depending on the running index j by a more flexible while statement allowing for more complicated, composite termination (or rather continuation) conditions. But this solution appears quite unattractive. It is admittedly less transparent than the program using a jump, in spite of the fact that the most frequently heard objection to the use of jumps is that they

obscure the program. The other objection is that the goto-less version (17) requires more comparisons and tests, and hence is less efficient.

The crux of the matter is that well-structured programs should not be obtained merely through the formalistic process of eliminating goto statements from programs that were conceived with that facility in mind, but that they must emerge from a proper design process. Two alternative solutions are presented here as illustrations.

In the first case, we abandon the notion that the program must necessarily be based on the statement

```
for i := 1 to n do                                   (18)
a[i] := the next suitable number
```

and consider the basic iteration step to consist of the generation of the next element of the sequence *x*, followed by the test for its acceptability.

```
i := 1;                                              (19)
while i ≤ n do
begin generate next x;
  assign it to a[i];
  check whether all a[j] are different from a[i];
  if so, proceed by incrementing i
end
```

This form makes it obvious that we are in trouble, if the sequence *x* should be such that *i* cannot be incremented any longer. Written in terms of our programming language, (19) becomes

```
i := 1;                                              (20)
while i ≤ n do
begin get(x);
  a[i] := x; j := 1;
  while a[j] ≠ x do j := j+1;
  if i = j then i := i+1
end
```

The second approach to this problem retains the basic concept of the solution as shown in (18). From there, its development is characterized by the following two snapshots:

```
for i := 1 to n do                                   (21)
  repeat generate the next x;
    check its acceptability
  until acceptable
```

```
for i := 1 to n do                                   (22)
  repeat get(x);
    a[i] := x; j := 1;
    while a[j] ≠ x do j := j+1;
  until i = j
```

In contrast to (20), this solution consists of *three* nested repetitions instead of only two, and therefore seems inferior at first sight. In fact, however, solution (22) turns out to be even more economical. The reason is that in (20) the test for continuation $i \leq n$ is actually unnecessary whenever $i \neq j$, since $i \neq j$ in this case implies $i < j$, and because *i* has not been altered since the last evaluation of $i \leq n$. Of course, program (22) is considerably more efficient than the original form with a jump (16).

This terminates our consideration of Example 3.

The question remains open, of course, whether jumps can *always* be avoided without disadvantage. I shall not venture to answer this question, particularly because the term "disadvantage" is sufficiently vague to admit many interpretations. But there is evidence of the existence of some characteristic and reasonably frequent situations which are expressed only with difficulty in terms of the language construct introduced above. A particular case is the *loop with exit(s) in the middle*. Lately it has led language designers to introduce specific constructs mirroring this case [12]. It turns out, however, that it is most difficult to find a satisfactory and linguistically suggestive formulation, and that sometimes solutions are invented that seem to merely replace the symbol goto by another word, such as exit or jump. For example, the construct

```
loop S1;                                             (23)
  exit if P;
  S2
end
```

with the parametric statements *S1*, *S2*, and the termination condition *P* might be adopted to express the program

```
L1: begin S1;                                        (24)
      if P then goto L2;
      S2; goto L1
L2: end
```

in a more concise and goto-free form.

Expressing (24) in terms of the basic repetitive statement forms does, indeed, often lead to undesirable complications, such as unnecessary reevaluation of conditions, or duplication of parts of the program, as is shown by the two proposals (25) and (26).

```
repeat S1;                                           (25)
  if ¬P then S2
until P
```

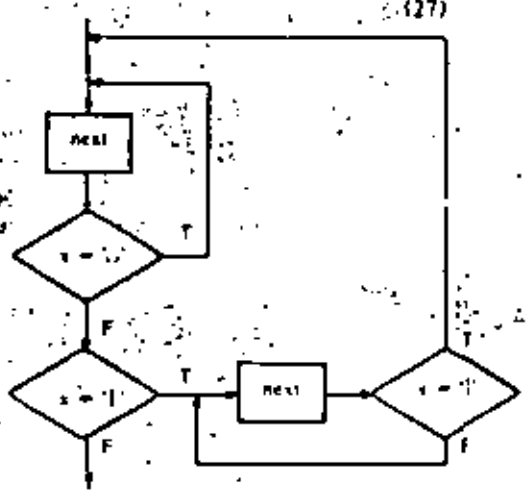
```
S1;                                                  (26)
while ¬P do
  begin S2; S1 end
```

The following, and last two, examples of problems are added to show that often the need for an *exit in the middle* construct is based on a preconceived notion rather than on a real necessity, and that sometimes an even better solution is found when sticking to the fundamental constructs.

Example 4: A scanner

The task is to construct a piece of program which, each time it is activated, scans an input sequence of characters, delivering as a result the next character, but skipping over blanks and over so-called comments. A comment is defined as any sequence of characters starting with a left bracket and ending with a right bracket.

This scanner could typically occur as part of a compiler. A common solution is indicated by the following flowchart (*next* denotes the operation of reading the next character and assigning it to the result variable *x*).



(27)

This program clearly exhibits the loop structure with exit in the middle, satisfying the one-entry-one-exit prerequisite. Instead of proposing a suggestive form for this construct in sequential language, however, let me tackle the posed problem in a different manner. Recognizing the main purpose of the program as being the reading of the next character, with the additional request for skipping over blanks and comments, I propose a first version as follows:

```
next;                                                    (28)
while x in ['␣', '['] do
  "skip blanks and comments"
```

The correctness of this program is easily established, assuming that the statement in quotes performs what it says, and nothing more. The definition of the while statement guarantees that the resulting value of *x* is neither a blank nor a comment; no matter in what way blanks and comments are skipped.

The refinement of the statement in quotes is guided by the fact that upon its initiation *x* is either a blank or an opening bracket.

```
if x = '␣' then next else                                (29)
  "skip comment"
```

where the last statement is expressed, with obvious reasoning, as

```
begin repeat next until x = ']';                         (30)
  next
end
```

Only knowledge about the expected *frequencies of occurrence* of individual characters can be a reason to choose another form of this program on the grounds of efficiency. For the sake of argument, let us assume that short sequences of blanks are particularly frequent and that, on the other hand, immediately adjacent comments are extremely rare. This leads us to an equally correct alternative form of (29), namely

```
"skip consecutive blanks, if any";                       (31)
"skip comment, if any"
```

The first of the two statements is readily expressed as

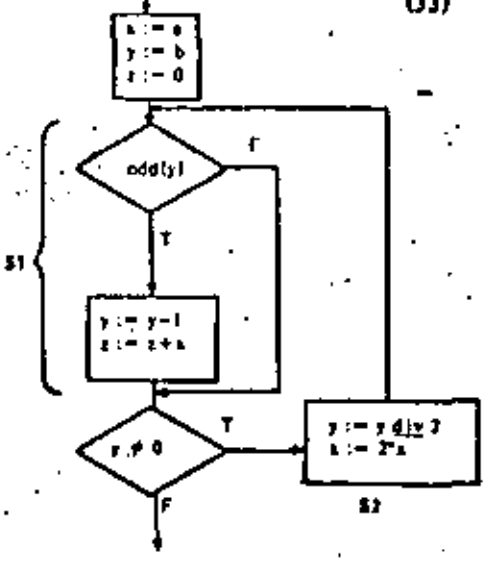
```
while x = '␣' do next                                    (32)
```

whereas the second is already elaborated in (30).

(33)

Example 5: Integer multiplication

Assume that we are to design a program to multiply two non-negative integers *a* and *b* with the use of addition, doubling, and halving only. Let the result be represented by a variable *z*. A well-known and efficient method is shown by the flowchart at right:



This program, once again, clearly exhibits the loop structure with exit in the middle, and therefore cannot be expressed as a single while statement. It is usually squeezed into the simple loop form by displacing the loop termination test, positioning it in front of statement $S1$. The program then obtains the well-known form

```
x := a; y := b; z := 0;                                   (34)
while y ≠ 0 do
  begin if odd(y) then
    begin y := y−1; z := z+x
    end;
    y := y div 2; x := 2*x
  end
```

This clearly does not change the effect of the program, because if $y = 0$ at entry to $S1$, then $S1$ has no effect and, in particular, leaves y unchanged; and if $y \neq 0$, then the only additional effect incurred by the modified version is on the auxiliary variables x and y in the case of $y = 1$. But this additional effect is quite undesirable, not so much because of the additional, superfluous, and useless computation, but because this operation may be harmful by causing overflow of the arithmetic unit. Should we therefore resort to the exit-in-the-middle version?

A different solution was shown to me by E.W. Dijkstra. He proposed to tackle the problem at its roots, instead of trying to remedy a preconceived proposal. The most obvious multiplication algorithm under the stated constraints is the following:

```
x := a; y := b; z := 0;                                   (35)
while y ≠ 0 do
  begin {y > 0 and x*y+z = a*b}
    y := y−1; z := x+z
  end
```

Before we start out trying to improve this version, we observe that at the outset of each repetition two conditions are satisfied.

1. $y > 0$ follows from the fact that y is a non-negative integer and not equal to zero.
2. $x*y+z = a*b$ is invariant under the two repeated assignments. (To verify this claim, substitute $y-1$ for y and $z+x$ for z: this yields $x*(y-1)+(z+x) = x*y+z = a*b$, i.e., the original equation.) At entry the equation is satisfied, since $z = 0$, $x = a$, $y = b$.

Note that the invariant equation combined with the negation of the continuation condition yields $(y = 0)$ and $(x*y+z = a*b)$, i.e., the desired result $z = a*b$.

If we now insert any statement at the place of the invariant which leaves the product $x*y$ unchanged, the result of the program will evidently remain the same. Such a statement is, e.g., the pair of assignments

```
y := y div 2; x := 2*x                                    (36)
```

under the condition that y is even. But if a relation is invariant over a statement, it remains so regardless of how often the statement is executed. This suggests the following, quite evidently correct, efficient, and elegant solution. It contains no exit-in-the-middle loop.

```
x := a; y := b; z := 0;                                   (37)
while y ≠ 0 do
  begin {y > 0 and x*y+z = a*b}
    while even(y) do
      begin y := y div 2; x := 2*x
      end;
    y := y−1; z := z+x
  end
```

So much for examples, whose purpose was to sketch and elucidate the basic ideas behind the methods of structured programming and stepwise refinement.

Conclusions

Skeptics will, of course, doubt that these methods represent any progress over the techniques of the old days — in fact, that they are *methods* at all. I can merely say that in my own experience, the new approach has improved my attitudes and abilities towards programming very considerably, and the experiences of others confirm this impression [10, 11]. A systematic, orderly, and transparent approach is mandatory in any sizable project nowadays, not only to make it work properly, but also to keep the programming cost within reasonable bounds. It is the very fact that computation has become very cheap in contrast with salaries of programmers, that squeezing the machines to yield their utmost in speed has become much less important than reliability, correctness, and organizational clarity. It is not only more urgent, but also much more costly to correct an efficient, but erroneous program, than to speed up a relatively slow, but correct program. In the past, the debugging phase has taken a ridiculously large percentage of the development cost in most large projects. The aim now is to eliminate the necessity of debugging by creating bug-free products in the first place. Doesn't this bring to mind the medical slogan "prevention is better than healing"!

The criticism has been voiced that the method of structured programming is in essence nothing more than programming by painstakingly avoiding the use of jumps (goto statements). One may, indeed, come to this conclusion by looking at the entire issue in the reverse direction. But in fact, the method of stepwise decomposition and refinement of the programming task automatically leads to goto-free programs; the absence of jumps is not the initial aim, but the final outcome of the exercise. The claim that structured programming was invented by proving that all programs can be formulated without goto statements is therefore based on a fundamental misunderstanding.

The question of whether jumps enter the picture or not is basically a matter of the level of decomposition or refinement to which the programming process is carried. Ultimately — that is in machine code — there can be no doubt about the presence of jump instructions. The moral of the story is that jumps must not be used in the initial conception of a general algorithmic strategy, and in fact should be delayed as long as possible. With today's state of technology, the introduction of jump instructions can be left to compilers of languages that offer adequate, judiciously chosen, disciplined structuring facilities.

One of the essential facilities for this purpose, besides conditional and repetitive statements, is the *recursive procedure*. In many cases it emerges as the natural formulation of a solution, such as, for instance, in most cases of backtracking algorithms. Hardly anywhere else can a natural, concise, and often self-explanatory solution be made more obscure and mystifying than by replacing its recursive formulation by one in terms of repetition and — well — jumps. This process should definitely be left to a compiler, as it concerns what is called *coding* rather than programming (code = system of symbols used in ciphers, secret messages, etc. [Webster]). Modern programming systems, however, offer efficient implementations of recursion, and thereby make "programming around recursion" a largely unnecessary exercise.

Whereas a teacher should not and must not pay attention to "percent issues" as to efficiency while explaining and exemplifying methods of composing well-structured programs, a professional programmer may well be forced to do so. He may sometimes find a dogma of sticking exclusively to a restricted set of program structuring schemas too much of a straight-jacket, and the temptation to break out too powerful. This will be the case as long as compilers are insufficiently sophisticated to take full advantage of disciplined structuring. Naturally, there will always be situations where a compiler is either denied the full information needed for successful code optimization, or where it would be unable to infer the necessary conditions. It is therefore entirely possible that in the future a more interactive mode of operation between compiler and programmer will emerge, at least for the very sophisticated professional. The purpose of this interaction would not, however, be the development of an algorithm or the debugging of a program, but rather its *improvement under invariance of correctness*.

The foregoing discussion also implies an answer to the question of whether structured programming in an unstructured language (such as FORTRAN) is possible. It is not. What is possible, however, is structured programming in a "higher level" language and subsequent hand-translation into the unstructured language. The corollary is that whereas this approach may be practicable with the almost superhuman discipline of a compiler, it is highly unsuited for *teaching* programming. Recognizing that there may be valid economic reasons for learning *coding* in, say, FORTRAN, the use of an unstructured language too teach *programming* — as the art of systematically developing algorithms — cannot no longer be defended in the context of computer science education. The lack of an adequate modern tool on the available computing facility is the only remaining excuse.

The last remark concerns an aspect of "structured programming" that has not been illuminated by the foregoing examples: structuring considerations of program and data are often closely related. Hence, it is only natural to subject also the specification of data to a process of stepwise refinement. Moreover, this process is naturally carried out simultaneously with the refinement of the program. A language must, therefore, not only offer program structuring facilities, but an adequate set of systematic data structuring facilities as well. An example of this direction of language development is the programming language PASCAL [12, 13]. The importance of this aspect of programming is particularly evident, as we recognize the data as the ultimate object of our interest: they represent the arguments and results of all computing processes. Only structure enables the programmer to recognize meaning in the computed information.

Acknowledgment

The author is grateful to P.J. Denning for kindly posing the problem treated in Example 3.

1. E.W. Dijkstra, "Some Meditations on Advanced Programming," *Proceedings of the 1962 IFIP Congress* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1963), pp. 535-38.
2. _____, "The Humble Programmer," *Communications of the ACM*, Vol. 15, No. 10 (October 1972), pp. 859-66.
3. _____, "Notes on Structured Programming," *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (New York: Academic Press, 1972).
4. P. Naur, B. Randell, and J.N. Buxton, eds., *Software Engineering, Concepts and Techniques, Proceedings of the NATO Conferences* (New York: Petrocelli/Charter, 1976).
5. N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, No. 4 (April 1971), pp. 221-27.
6. _____, *Systematic Programming* (Englewood Cliffs, N.J.: Prentice-Hall, 1973).
7. P. Naur, "Proof of Algorithms by General Snapshots," *BIT*, Vol. 6, No. 4 (1966), pp. 310-16.
8. C.A.R. Hoare, "An Axiomatic Approach to Computer Programming," *Communications of the ACM*, Vol. 12, No. 10 (October 1969), pp. 576-80, 583.
9. W.A. Wulf, "Programming Without the GOTO," *Proceedings of the 1971 IFIP Congress*, Vol. 1 (Amsterdam, The Netherlands: North-Holland Publishing Co., 1972), pp. 408-13.
10. F.T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, Vol. 11, No. 1 (January 1972), pp. 56-73.
11. U. Ammann, "The Method of Structured Programming Applied to the Development of a Compilet," *Proceedings of the 1973 International Computing Symposium* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974), pp. 93-100.
12. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, Vol. 1, No. 1 (1971), pp. 35-63.
13. K. Jensen and N. Wirth, "PASCAL — User Manual and Report," *Lecture Notes in Computer Science*, Vol. 18 (New York: Springer-Verlag, 1974).

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

ARTICULOS CORRESPONDIENTES AL TEMA III

DISEÑO ESTRUCTURADO

Ing. Alejandro Acosta

ABRIL, 1982

Module connection analysis - A tool for scheduling software debugging activities	Frederick N. Haney
Order and Discipline: Benefits of structured Techniques	David S. Iwahashi
Structured Systems Analysis: A Technique to Define Business Requirements	Kathleen S. Mendes
HIPO and integrated program design	J.F. Stay
Conversion of unstructured flow diagrams to structured form.	M.H. Williams and H.L. Ossher
The second Structured Revolution	Edward Yourdon
Composite design facilities of six Programming languages	G. J. Myers
A summary of progress toward proving program correctness	T.A. Linden
Improved software development through projet management	Robert B. Fireworker and Leonard J. Bogner, Jr.
Structured design	W.P. Stevens, G.L. Myers, and L.L. Constantine
Is this Really Necessary? A First Look at Design Techniques	Gregg Williams

Module connection analysis—A tool for scheduling software debugging activities

by FREDERICK M. HANEY

Xerox Corporation
El Segundo, California

INTRODUCTION

The largest challenge facing software engineers today is to find ways to deliver large systems on schedule. Past experience obviously indicates that this is not a well-understood problem. The development costs and schedules for many large systems have exceeded the most conservative, contingency-laden estimates that anyone dared to make. Why has this happened? There must be a plethora of explanations and excuses, but I think H. R. J. Grosch identified the common denominator in his article, "Why MAC, MIS and ABM will never fly."[1] Grosch's observation is essentially that for some large systems the problem to be solved and the system designed to solve it are in such constant flux that stability is never achieved. Even for some systems that are flying today, it is obvious that they came precariously close to this unstable, "critical mass" state.

It is my feeling that our most significant problem has been gross underestimation of the effort required to *change* (either for purposes of debugging or adding function) a large, complex system. Most existing systems spent several years in a state of gradual, painfully slow transition toward a releasable product. This transition was only partially anticipated and almost entirely unstructured; it was a time for putting out fires with little expectation about where the next one would occur.

The difficulties of stabilizing large systems are universal enough that our experience has resulted in several improved methods for estimating projects. Rules-of-thumb like "10 lines of code per man day" once sounded like extremely conservative allowance for the complexities of system integration and testing. J. D. Aron[2] has described a relatively elaborate technique for estimating total effort for large projects. Aron's technique is based on the estimated amount of code for a project and empirically observed distributions of various kinds of effort such as design, coding, module testing, etc. More recently Belady and Lehman described a mathematical model for the "meta-dynamics of systems in growth."[3] These schemes provide useful insights into the difficulties of designing and implementing large systems.

Even with these improved estimation techniques, however, we still face the threat of long periods of unstructured post-integration putting out of fires. We may know better how long this "final" debugging will take, but we are still at a loss to predict what resources will be required or what specific activities will take place. If we predict an 18 month period for "final testing," will management buy it? How can we peer into this hazy contingency portion of a schedule and predict in greater detail where bugs will occur, who will be needed to fix them, elapsed time between internal releases, etc.? Belady and Lehman suggest the need for a "micro-model" for system activities; i.e., a model based on internal, structural aspects of a system. This is essentially the objective of this paper. In the following sections, we will develop a very simple, but useful, technique for modeling the "stabilization" of a large system as a function of its internal structure.

The concrete result described in this paper is a simple matrix formula which serves as a useful *model* for the "rippling" effect of changes in a system. The real emphasis is on the use of the formula as a model; i.e., as an aid to understanding. The formula can certainly be used to obtain numeric estimates for specific systems, but its greater value is that it helps to *explain*, in terms of system structure and complexity, why the process of changing a system is generally more involved than our intuition leads us to believe.

The technique described here, called *Module Connection Analysis*, is based on the idea that every module pair (may be replaced by subsystem, component, or any other classification) of a system has a finite (possibly 0)

probability that a change in one module will necessitate a change in any other module. By interpreting these probabilities and applying elementary matrix algebra, we can derive formulae for estimating the total number of "changes" required to stabilize a system and the staging of internal releases. The total number of changes, by module, is given by

$$A \times (I-P)^{-1},$$

where A is a row vector representing the initial changes per module, P is a matrix such that Pij is the probability that a change in module i necessitates a change in module j, and I is the $n \times n$ identity matrix. The number of changes required for each "internal release" is given by AP^k, $K=0, 1, \ldots$, or by

$$A \times (I-P)^{-1} \times Uk, \qquad k=1, 2, \ldots n,$$
$$Uk = (0, \ldots, 1, \ldots 0)$$
$$\uparrow$$
$$k \text{ th element}$$

depending upon the release strategy. The derivations of these formulae are presented in the following section.

Module connection analysis is useful primarily as a tool for augmenting a designer's quantitative understanding of his problem. It produces quantitative estimates of the effects of module interconnections, an area in which intuitive judgment is generally inadequate.

THEORY OF MODULE CONNECTIONS

As a basis for our analysis, we postulate several characteristics of a system:

- A system is hierarchical in structure. It may consist of subsystems, which contain components, which contain modules or it may be completely general having n different levels of composition where an object at any level is composed of objects at the next lower level.
- At any level of the hierarchy, there may be some interdependence between any two parts of the system.
- If we view a system as a collection of modules (or, whatever object resides at the lowest hierarchical level), then the various interdependencies are manifested in terms of dependencies between all pairs of modules.

By dependence here, we mean that a change in one module may necessitate a change in the other. The fundamental axiom of module connection analysis is that intermodule connections are the essential culprit in elongated schedules. That a change in one module creates the necessity for changes in other modules, and these changes create others, and so on. Later, we will see that perfectly harmless-looking assumptions lead easily to sums like hundreds of changes required as a result of a single initial change. (The notions of hierarchy, interconnection, etc., used here are described at length in Reference 4.)

If we assume that a system consists of n "modules," then there are n^2 pairwise relationships of the form—

Pij = Probability that a change in module i necessitates a change in module j.

In the following, the letter "P" denotes the $n \times n$ matrix with elements pij. Furthermore, with each module i, there is associated a number Ai of changes that must be made in module i upon integration with the system. (Ai is approximately the number of bugs that show up in module i when it is integrated with the system.) If we let A denote a row vector with elements Ai, then we have the following:

A = total changes, by module, required at integration time, or at *internal release* 0.

AP = total changes required, by module, as a result of changes made in release 0, or total changes for *internal release* 1.

(Internal release $n+1$ is, roughly, a version of the system containing fixes for all first-order problems in internal release n.)

Now we observe that the i, jth element of P^2 is

$$\sum_{k=1}^{n} Pik\ Pkj,$$

which represents the sum of probabilities that a change in module i is propagated to module k and then to module j. Hence, the i, jth element of P^2 is the "two-step" probability that a change in module i propagates to module j. Or, AP^2 is the number of changes required in *internal release* 2.

The generalization is now obvious. The number of changes required in internal release k is given by AP^k and the total number of changes, T, is given by

$$T = A(I+P+P^2+P^3+\cdots).$$

Now we are interested to know whether or not the matrix power series in P converges; clearly, if it does not our system will never stabilize. To establish con-

ergence of the power series, we appeal to matrix algebra (see Reference 5, for example) which tells us that the above series converges whenever the eigenvalues of P are less than 1 in absolute value. If this is the case, then the series converges and

$$T = A(I-P)^{-1}, \quad \text{where } I \text{ is the } n \times n \text{ identity matrix.}$$

We now have an extremely simple way to estimate the total number of changes required to stabilize a system as a linear function of a set of initial changes, A. Moreover, the number of changes at each release is given by the elements of AI, AP, AP^2, etc.

ESTIMATING TOTAL DEBUGGING EFFORT FOR A SYSTEM

The above theory suggests a simple procedure for estimating the total number of changes required to stabilize a system. The procedure is as follows:

(1) For each module pair, i, j, estimate the probability that a change in module i will force a change in module j. These estimates constitute the probability matrix P.
(2) From the vector A by estimating for each module i the number of "zero-order" changes, or changes required at integration time.
(3) Compute the total number of changes, by module:
$$T = A(I-P)^{-1}.$$
(4) Sum the elements of the column vector T to obtain the total number of changes, N.
(5) Make a simple extrapolation to "total time" based on past experience and knowledge of the environment. If past experience suggests a "fix" rate of d per week, then the total number of weeks required is N/d.

Hence if we have some estimate for the initial correctness (or "bugginess") of a system and for the inter-module connectivity (the probabilities), then we can easily obtain an estimate for the *total* number of changes that will be required in debugging the system. The formula is a simple one in matrix notation, but the fact that we are dealing with matrices probably explains the failure of our intuition in understanding debugging problems.

In the following sections, we will show how the above formula can be used to aid our understanding of other aspects of the debugging process.

STAGING INTERNAL RELEASES

There are various strategies for tracking down bugs in a complex system. The most obvious are: (1) fix all bugs in one selected module and chase down *all* side effects, or, (2) fix all "first-order" bugs in each module, then fix all "second-order" bugs, and so on. The module connection model can aid in predicting release intervals for either approach.

For strategy (1) (one module at a time), the number of changes required to stabilize module i, given Ai initial changes, is given by

$$(p, \ldots, Ai, \ldots, 0)(I-P)^{-1}$$

The product is a row vector with elements corresponding to the number of changes that must be made in each module as a result of the original changes. The total number of changes required to stabilize this one release is given by

$$Ai \sum_{k=1}^{n} Xik,$$

where the Xik are elements of $(I-P)^{-1}$. This strategy, then results in n internal releases where the time for release i is

$$Ai(\max_k Xik) \times (\text{time required per change})$$

and the total debug time after integration is

$$\sum_i (Ai \max_k Xik) \times (\text{time required per change})$$

With the second debugging strategy (make all "first-order" changes, then all "second-order" changes, etc.), the number of changes in the kth release is given by AP^k. That is, AI^n is a row vector with elements corresponding to the number of changes in each module for release k. The time required for release k is approximately

$$\max(AP^k) \times \text{time required per change.}$$

To determine the total number of releases for this strategy, we must examine A, AP, AP^2, until the number of changes AP^s in release s is small enough that the system is releasable. The total time for this strategy, then, is

$$\sum_{k=0}^{s} \max AP^k \times \text{time required per change.}$$

It is worth noting that both of the debug strategies described above evidence a "critical path" effect. The total time in each case is a sum of maximum times for each release. This effect corresponds to the well-known fact that debugging is generally a highly sequential

process with only minor possibilities for making many fixes in parallel. This fact, coupled with the "amplification" of changes caused by rippling effects, certainly accounts for a large portion of many schedule slips.

REFINING THE INITIAL ESTIMATES

Module connection analysis is proposed as a tool for aiding designers and implementors. More than anything else, it is a rationale for making detailed quantitative estimates for what is generally called "contingency." Now, we must ask, "As a project progresses, how can we take advantage of actual experience to refine the initial estimates?" The module connection model is based on two objects: A, the vector of initial changes; and P, the matrix of connection probabilities between the modules. Both A and P can be revised simply as live data become available.

As each module i is integrated into the system, the number Ai of initial changes becomes apparent.

Using updated values for the vector A, we can recompute the expected total number of changes and the revised release strategy.

The elements, Pij, of the matrix P can be revised periodically if sufficient data is kept on changes, their causes, and their after-effects. One simple way to do this is to keep a record for each module as follows:

Module i		
description of change	caused by which module?	other modules affected

After a relatively large sample of data is available, the above forms can be used to revise P as follows:

$$Pij = \frac{\text{number of changes in } j \text{ caused by } i}{\text{total changes made to } i}$$

The revised matrix P can be used to revise earlier estimates for total effort and release strategies.

AN EXAMPLE OF MODULE CONNECTION ANALYSIS

The following example is based on the Xerox Universal Timesharing System. Eighteen actual subsystems are used as "modules." Estimates for connection probabilities and initial changes are made in the same way that they would be made for a new system, except that some experience and "feel" for the system were used to obtain realistic numbers. (Thanks to G. E. Bryan, Xerox Corporation, for helping to construct this example.)

.2	.1	0	0	0	.1	0	.1	0	.1	.1	.1	0	0	0	.1	0	0
0	.2	0	0	.1	.1	.1	0	0	0	0	0	.1	.1	.1	0	.1	0
0	0	.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	.1	0	.2	0	.1	.1	.1	0	0	0	0	0	0	.1	0	.1	0
.1	0	0	0	.4	.1	.1	.1	0	0	0	0	0	0	0	0	.1	0
.1	0	0	0	0	.3	.1	0	0	.1	0	0	0	.1	0	0	.1	0
.1	0	0	.1	.2	.1	.3	.1	0	.1	0	0	0	.1	0	.1	.1	0
.1	.1	0	.1	.2	0	.1	.4	0	.1	0	0	0	.1	0	0	0	.1
0	0	0	0	0	0	0	0	.1	0	0	0	0	0	0	0	0	0
.1	0	0	0	0	.1	.1	.1	0	.4	.2	.1	.2	.1	.1	.1	.1	.1
.1	0	0	.1	0	0	0	0	0	.2	.3	.1	0	0	0	0	0	0
.2	0	0	0	0	.1	0	0	0	0	.2	.3	0	0	.1	.1	0	0
.1	.1	0	0	0	.1	.1	.1	0	.2	.1	0	.3	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.2	0
0	0	0	0	.1	0	.1	0	0	.1	0	0	0	0	0	0	0	.3

Figure 1—Probability connection matrix, P

The 18×18 probability connection matrix for this example is given in Figure 1. The matrix is relatively sparse; moreover, most of the nonzero elements have a value of .1. Most the larger elements lie on the diagonal

INITIAL AND FINAL CHANGES

Module	Initial Changes	Total Required Changes
1	2	241.817
2	8	100.716
3	4	4.44444
4	6	98.1284
5	28	248.635
6	12	230.970
7	8	228.951
8	28	257.467
9	4	4.44444
10	8	318.754
11	40	238.609
12	12	131.311
13	16	128.318
14	12	157.108
15	12	96.1139
16	28	150.104
17	28	188.295
18	40	139.460
TOTALS	296	2963.85

Figure 2

corresponding to the fact that the subsystems are relatively large so that the probability of ripple within a subsystem is relatively large.

The total number of changes required in each module are given in Figure 2. It is interesting to note which modules require the most changes and to observe that six modules account for 50 percent of the changes.

Figure 3 illustrates the one-release-per-module debug strategy. That is, we repair one module and all side effects, then another module, and so on.

This strategy is rather erratic since the time between releases, which is determined by the maximum number of fixes in one module, ranges from 4 to 95 indiscriminately. If we adopt this strategy, we may want to select the worst module first and continue using the worst module at each step. We will see, however, that this strategy is far from optimal because it does not take maximum advantage of opportunities to make fixes in parallel.

ONE RELEASE PER MODULE

Release	Maximum Changes in One Module
1	4.41764
2	11.8019
3	4.44444
4	6.84029
5	67.8994
6	24.7185
7	20.3720
8	85.8099
9	4.44444
10	35.2978
11	95.2147
12	22.5608
13	39.7013
14	15.0000
15	15.0000
16	35.0000
17	35.0000
18	66.5554
"CRITICAL PATH" TOTAL	592.138

Figure 3

A more effective release strategy is illustrated in Figure 4. This strategy assumes all first-order changes in release 1, all second order changes in release 2, etc. Figure 4 shows, for each release, the maximum number of changes in one module and the total number of changes. The reader who has worked on a large system will, no doubt, recognize the painfully slow convergence pattern. In this case, the system is assumed to be ready for external release when the "maximum changes per module" becomes less than one.

If we assume the "critical path" changes are made at

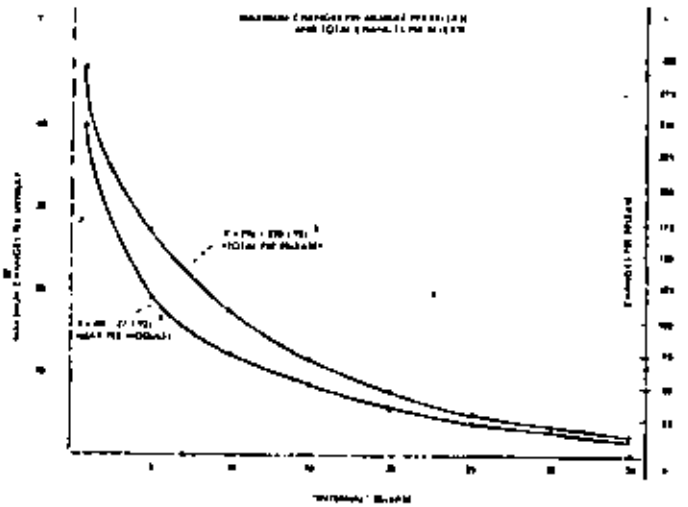


Figure 4—"Internal" release

an average rate of about 1 per day, then Figure 4 is fairly representative of experience with the first release of UTS. The total number of changes on the "critical path" is 338, so that approximately 15 months would

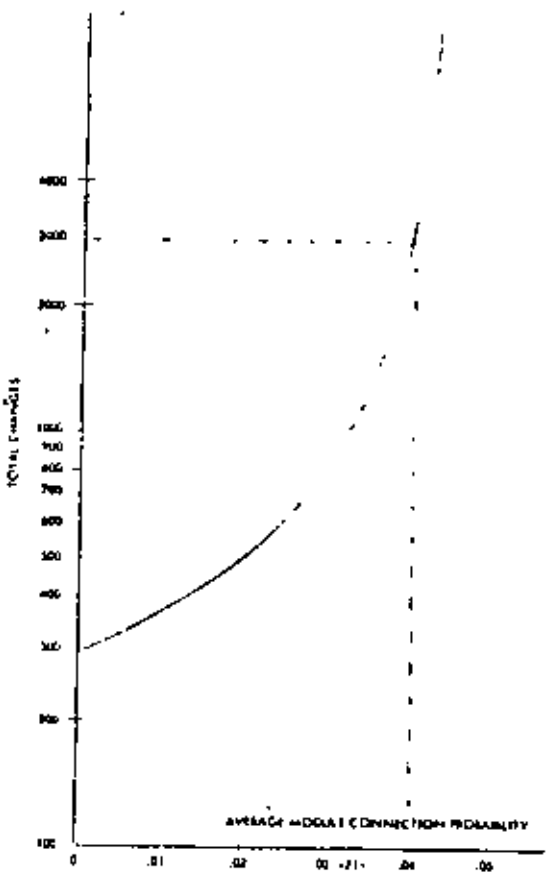


Figure 5—Total changes as a function of "average connection probability"

be required to stabilize the system for the first external release.

To conclude this example, let us take a brief look at the relationship between "total changes" and the probability of intermodule connection. The probabilities in the connection matrix above have an average value of approximately .04. What is the result if we assume the same relative distribution of probabilities in the matrix, but reduce the average by dividing each element by a constant?

Figure 5 shows the total number of changes as a function of "average probability of module connection" under the above assumption. This curve shows that our example is precariously close to "critical mass" and that any small improvement in the connection probabilities results in significant payoff.

OTHER APPLICATIONS OF MODULE CONNECTION ANALYSIS

The value of module connection analysis is its simplicity. The computations can be performed easily by a small (less than 50 lines) program written in APL, BASIC, or whatever language is available. Used on-line, the technique is useful for experimenting with various design approaches, implementation strategies, etc. Three examples of this use of the model are described below:

Estimating new work

If the designers, or managers, of a system have kept detailed records of the module-module changes in the system (as described above), then the matrix P is a reliable estimator of the "ripple factor" for the system. It can be used to predict, and stage, the effort to stabilize the system after any set of changes. If we postulate a major improvement release of the system, then we can assume, for example, that the new program code falls into two categories: (1) independent code particular to a new function and, (2) code that necessitates changes in an existing module. By estimating the number of changes, bi, to each module i, we can estimate the total number of changes to restabilize the system:

$$\text{Total changes} = (b_1, b_2, \ldots, b_n)(I-P)^{-1}.$$

The previously described computations can be used to estimate release intervals and total time for the improvement release.

To be more realistic, it may be useful in the above computation to use $bi+ei$ as the estimated changes in the module, where ei represents the number of changes required in module i by previous activity.

Evaluating design approaches

The best time to guarantee success of a system development effort is in the early design stages when architecture of the system is still variable. There is much to be gained by selecting an appropriate "decomposition" (see Reference 4). of the system into subsystems, components, etc. During this stage of a project, module connection analysis is a useful tool for evaluating various decompositions, interfacing techniques, etc. It is a simple, *quantitative* way of estimating the *modularity* of a system, the ever-present objective that no one knows exactly how to achieve. By fixing some of his assumptions about intermodule connections, a designer can experiment with various system organizations to determine which are the least likely to achieve "critical mass."

Evaluating implementation approaches

The reader who performs some simple experiments with the formulas described here is likely to be very surprised at the results. Even an extremely sparse connection matrix with very low probabilities can result [examine $(I-P)^{-1}$] in very large "ripple factors." It is also interesting to experiment with small perturbations in the connection matrix and observe the profound effect they can have on the "ripple factor." One becomes convinced more than ever before that it is necessary to minimize connections between modules, localize changes, and simplify the process of making changes.

The most impressive gains come from minimizing the probabilities of intermodule propagation of changes. A reduction of the average probability by as little as 5 or 10 percent can cause a significant reduction in the "ripple factor." Additional improvement can result from improvements in techniques for making changes. The total debug time is essentially linear with respect to the time required to make a change, but the multiplier (total number of changes) can be so large that any reduction in the time-per-change results in enormous savings.

Module connection techniques are extremely useful in estimating the value of various implementation techniques and strategies. How are the module connection probabilities changed if we use a high-level implementation language? How much easier will it be to

make changes? How much will we save, if any, by doing elaborate environment simulation and testing of each module before it is integrated with the system? Module connection analysis is a valuable augmentation of intuition in these areas and can be useful for generating cost justifications for approaches that result in significant savings.

CONCLUSION

The objective of this paper has been to describe a simple model for the effect of "rippling changes" in a large system. The model can be used to estimate the number of changes and a release strategy for stabilizing a system given any set of initial changes. The model can be criticized for being simplistic, yet it seems to describe the *essence* of the problem of stabilizing a system. It is clear, to the author at least, that experimentation with the module connection model could have prevented a significant portion of the schedule delay that occurred for many large systems.

REFERENCES

1 H R J GROSCH
Why MAC, MIS, and ABM won't fly
Datamation 17 Nov 1 1971 pp 71-72

2 J D ARON
Estimating resources for large programming systems
Software Engineering Techniques J N Buxton and B Randell (eds) April 1970

3 L A BELADY M M LEHMAN
Programming system dynamics or the meta-dynamics of systems in maintenance and growth
Research Report IBM Thomas J Watson Research Center Yorktown Heights New York July 1971

4 C ALEXANDER
Notes on the synthesis of form
Cambridge Mass Harvard University Press 1964

5 M MARCUS
Basic theorems in matrix theory
National Bureau of Standards Applied Mathematics Series #57 U S Government Printing Office January 1960

The utility and applicability of structured techniques must be recognized before they can be successfully implemented.

ORDER AND DISCIPLINE: BENEFITS OF STRUCTURED TECHNIQUES

by David S. Iwahashi

In the last few years, many articles have been published expounding the virtues of structured programming concepts and techniques. These techniques reportedly have improved software manageability, productivity, quality, and maintainability. However, in spite of these claims, large software development efforts are still plagued with the classic programming problems:

- Inaccurate, overly optimistic percentage-complete estimates. Programmers have been characterized as notoriously unreliable in providing accurate completion estimates, as in the "95% complete" syndrome.
- Uneven productivity throughout the development effort. Disarray, overtime, and waived standards increase as the development approaches the due date.
- Excessively complex systems design. The result is fragile software that is difficult to maintain and modify.
- Inadequate documentation and a lack of appropriate standards and insight into the needs of the user.
- Inaccurate instruction count estimates. Deriving cost estimates entirely from software instruction count estimates is not a reliable technique; cost overruns may frequently result.

Theories and textbook methodology are often inadequate or unrealistic. Software systems are developed in unique environments with diverse characteristics. Furthermore, the human element is a major variable, with a complexity that is accentuated in large-scale software efforts. The art of effective large-scale software development requires insight, imagination, and creativity.

As the chief programmer on a large-scale effort, I attempted to maintain a perspective on problems and appropriate solutions. A technically sound system was paramount. Fundamental software and management control techniques were considered simultaneously in order to facilitate comprehension between programmers and nonprogrammers. The basic techniques and objectives were structured design to simplify system complexity and understanding, status and control techniques to provide progress visibility and realism, checkout milestones to impose steady productivity, and standards and guidelines to enhance code and documentation quality.

Here, we will discuss how several structured programming concepts were incorporated into a large-scale scientific software development effort (15 programmers, 100K lines of code, 200 man-months). This scientific software development effort consisted of two separate but dependent programs—the display program and the batch program.

The display program consists of approximately 19 separate displays that provide the user with the capability to generate and update controls for a real-time system; to specify data sets, schedule and optimize data; and to generate and update control files and list files. This program can read 12 different types of control and reference files and write nine. The approximate program size is 77 overlays and suboverlays, 390 subroutines, 60K total FORTRAN source instructions, 11K total machine language source instructions, and 117K total cards.

The batch program has the capability to generate, update, and list nine reference files in a batch mode; these reference files are used by the display program. The approximate size of the program is 42 overlays and suboverlays, 366 subroutines, 20K total FORTRAN source instructions, 4K total machine language source instructions, and 42K total cards. (Core restriction requirement for both programs produced numerous overlays and suboverlays.)

The programs were developed on Control Data 6000 series equipment. Initial requirements had been established during a nine-month study effort. Program development required approximately seven months for requirements generation, preliminary design, and detail design; about 11 months were needed for code, checkout, and testing.

STAFF DIVIDED IN THREE PARTS

The software development staff was organized into programming, software integration, and system engineering groups. This organizational structure had been in use for approximately nine years, producing similar special-purpose scientific software. These software products and development efforts met requirements, were delivered relatively on schedule (a few months late at worst), and required extensive programmer overtime (averaging 11% at worst), especially during the final phase of checkout. The efforts also occasionally overran cost estimates slightly (+10% at worst), and were relatively error free.

Several structured programming techniques were implemented on these previous efforts, including top-down code development, utilization of intermediate programming language (IPL), code indentation, GOTO-less programs, and code reviews.

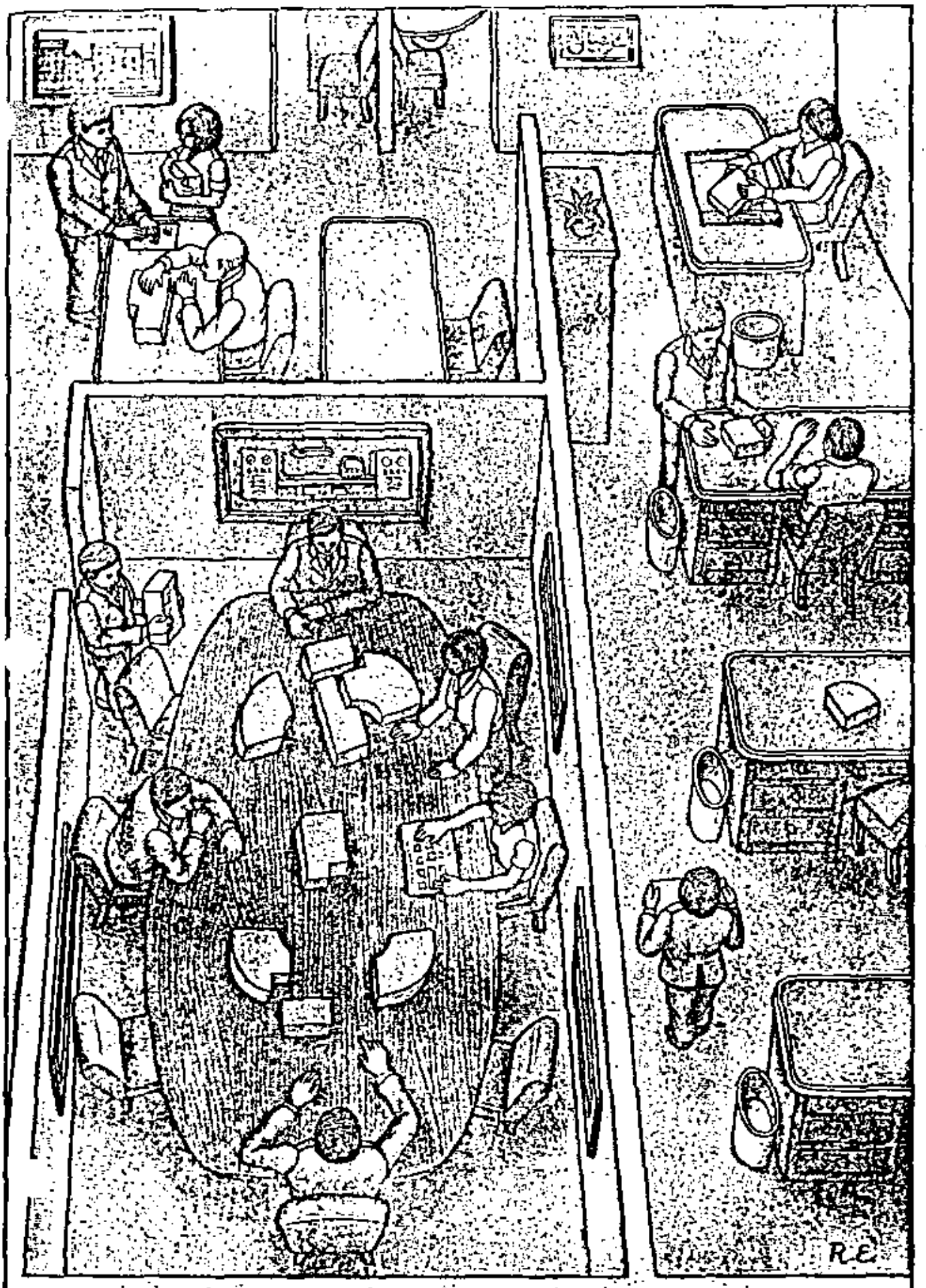
These earlier structured efforts experienced some software development problems. The indented code was difficult to update and maintain, reviews tended to be cursory, and extensive use of GOTO-less code took longer to execute and was difficult to follow. As a result, the maintenance programmers complained loudly about structured programming.

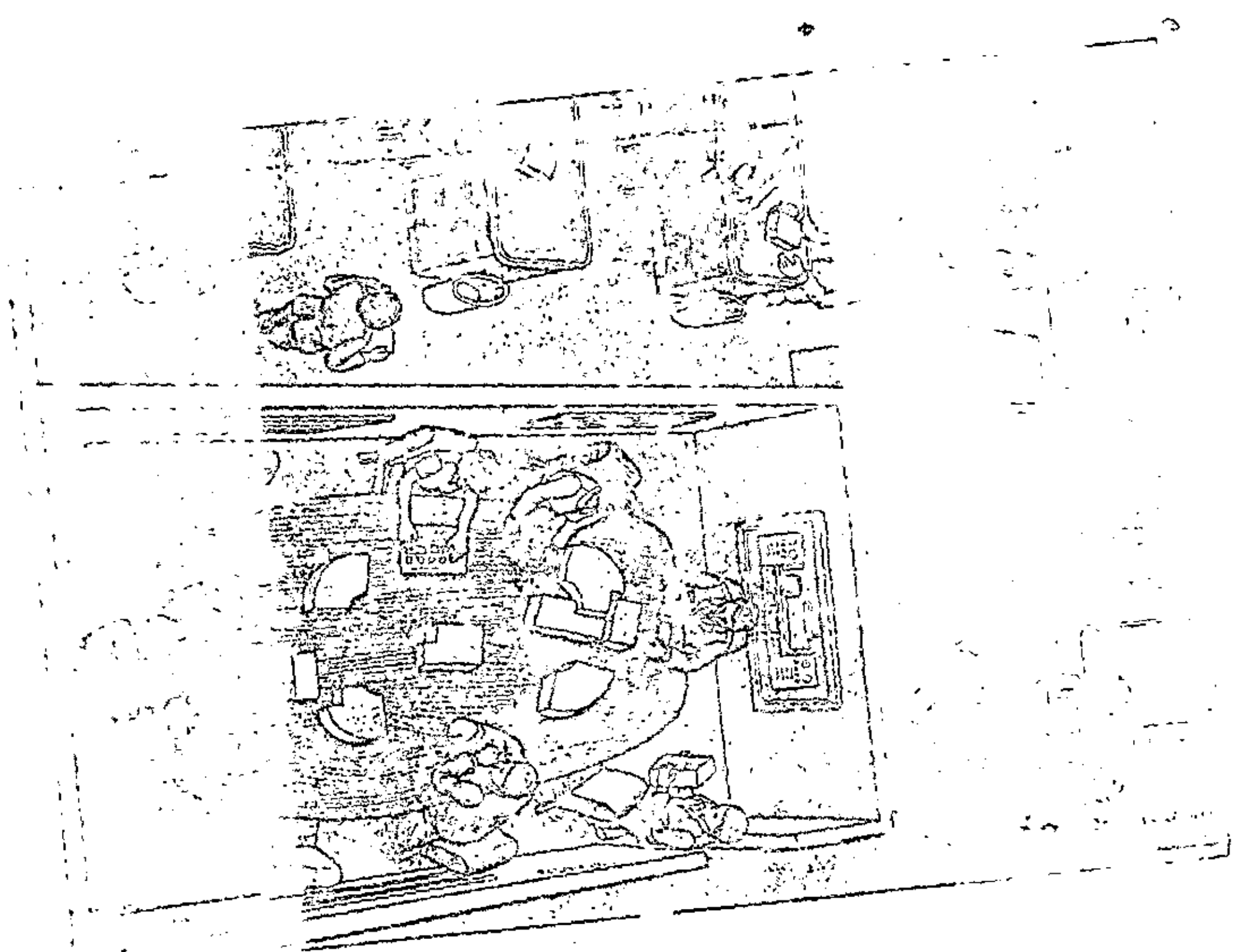
The organization, staffing, and responsibility on this software development effort were as follows:

Chief Programmer. The chief programmer performs a specific set of functions, and is not a position in name only. The chief programmer need not be a superprogrammer. On this effort, it was more important that the chief programmer be a software system designer, understand and appreciate software development problems, and have some imagination and insight for effective task assignments and control. The day-to-day programmer interface and guidance required for software design reviews and technical decisions indicate that the chief programmer should come from the software pits.

The chief programmer had overall responsibility for the entire development. A thorough and comprehensive understanding of all technical requirements was mandatory. The chief programmer had extensive, detailed experience (10 years) with the computer system, customer, and development procedures.

It is my opinion that the chief programmer can effectively direct, control, and offer technical guidance to only six to eight team members performing unique


الرجال
RE



Ultimately, it is the individuals that use the techniques who govern the final outcome of a development effort.

tasks. If there are more subordinates, some capable assistants are required. On this effort, software integration personnel provided the necessary assistance through technical guidance on the complex algorithms, intermediate program checkout and verification, and documentation reviews.

Assistant Chief Programmer. The assistant aids the chief programmer and handles lower-level design decisions. An assistant is necessary on large software development efforts to review and critique technical decisions, follow up in problem areas, assist with external communication and interfaces, train and brief new personnel, and minimize stopgap programming. The assistant also gets valuable training and experience, which insures the project against the loss of the chief programmer.

Unfortunately, this development effort did not have an assistant because programming personnel were unavailable. As a result, the programmers and software integration personnel were required to work harder.

Programmers. The programmers had responsibility for detailed software design, design documentation, development of code, and program checkout. Up to 15 programmers were required to complete this development effort. Programmer experience ranged from one to 19 years and every programmer had some previous experience on the applicable computer equipment. Individual programmers differed greatly in ability and experience. The number of years of software experience does not measure a programmer's applicable experience, true ability, or overall professionalism. It is necessary to correlate an individual programmer's experience, capability, and track record with the complexity of the specific task.

Task complexity ranged from straightforward data manipulation (75% of resultant code) to complex mathematical scheduling and optimization algorithms. The hierarchical diagram and structured design provide a means of better understanding complexity and help provide a basis for appropriate personnel assignments. Attempts were made to assign tasks according to programmer experience, expertise, track record, and individual preference. Although no generalization should be derived, the following are interesting observations:

- The less experienced programmers tended to do checkout superficially and have more discrepancies.
- Competent programmers successfully completed tasks regardless of complexity.
- The less productive programmers tended to govern on-time product delivery.

All major tasks were sized into manageable subtasks; major subtasks were further subdivided so that each component required approximately three to six months' effort. Manageable subtasks made it easier to accommodate schedule slippages and personnel attrition with reassignments to supplement development progress.

Librarian. The librarian's function was to keep track of program versions and files, and verify program updates throughout the development effort. This was an important responsibility and a key reason the development made steady progress. The most recent program version and listing were always available for checkout; no time was lost due to improper updates being used.

System Programmer. This programmer was responsible for computer system problems and anomalies. Since this effort was developed on an established computer and operating system, the support was on a problem-solving basis only. On new large-scale developmental computer systems, it would be advisable to have a resident system programmer available to handle the day-to-day problems encountered during software development.

Software Integration Group. This group was responsible for the coordination of the development effort, algorithm support, intermediate program checkout and user's manuals. The integration group provided a useful interface buffer between the programming group and the system engineering group. The complexity of the mathematical scheduling and optimization algorithms necessitated one member of the integration group be assigned to follow, monitor, checkout, and verify each algorithm.

System Engineering Group. This group was responsible for the software requirements, formal software testing, and customer interface.

SOFTWARE REQUIREMENTS DOCUMENT

Nine months of earlier engineering effort culminated in a study report that identified existing system limitations and proposed improvements, a design approach, and design requirements. This report contained numerous separate engineering desires and concepts, as well as ambiguities and uncertainties.

The software requirements document was written by the system engineering group. Even though the study report provided initial requirements, additional effort had to be expended to rewrite the requirements to provide logical and functional specifications rather than detailed algorithm and implementation design specifics. Software development efforts frequently begin with a feasibility analysis. A structured top-down approach to these studies and reports could significantly enhance a structured implementation.

Preliminary Design. The overall problem and the computer system and resources must be totally understood before a feasible software solution can be designed. The initial design was in hierarchical form and indicated how the problem could be solved in an orderly, well-structured manner. The initial diagrams were completed by the chief programmer in a one-month period, prior to establishing the programming team. These diagrams provided program organization in a top-down, stepwise refinement manner, as well as verification that all requirements were included (diagram contained study report and requirement document paragraph numbers for cross-reference); an outline for preliminary and detail design documentation; identification of subprogram size and efforts for personnel assignments; a clear delineation of responsibility; new personnel indoctrination/briefing information; identification of areas of concentration; identification of common utility and library routines; minimization of intraprogram communication (key factor in simplifying programming tasks by reducing interfaces).

The initial hierarchical diagrams were of sufficient detail (10 to 12 levels for each function) to show that the problem could be solved, and that all requirements were incorporated. These diagrams were not superficial; rather, they had sufficient depth to identify the complexity and magnitude of the subprograms. The key design features were to establish communication and control high within the hierarchy, standardize and minimize communication between hierarchical components, and design manageable components which could be independently verified.

Good, detailed hierarchical diagrams simplify subsequent assignments of tasks and responsibilities, as well as simplify control. Hierarchical diagrams cannot replace flow diagrams. Flow diagrams identify sequences, controls, and data flow, and are useful in describing the details of complex algorithms.

The hierarchical diagrams were the entire basis of the 450-page preliminary design documentation, and took about three months to develop. The chief programmer developed the overall docu-

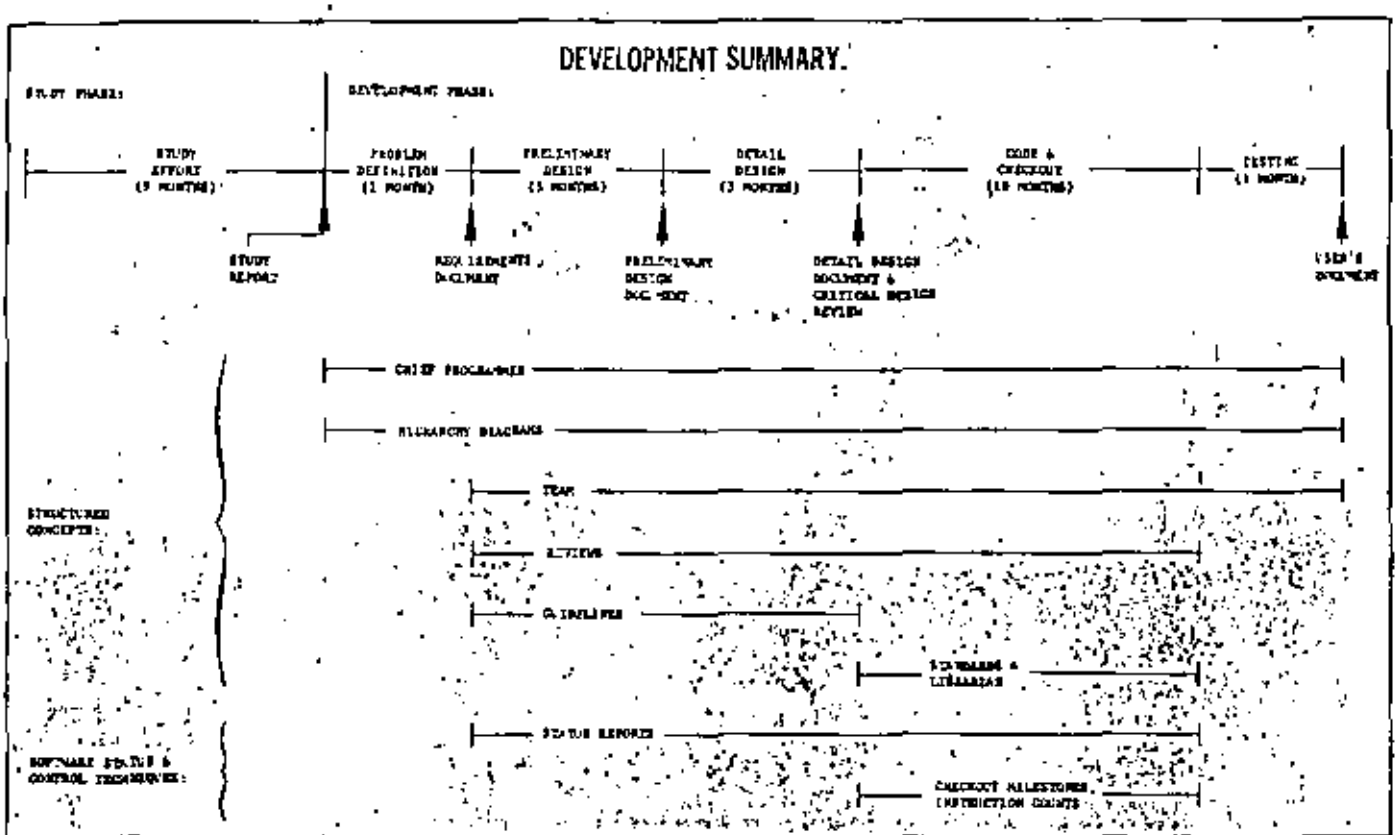


Fig. 1.

mentation outline and preliminary design guidelines. The basic documentation guideline was that the preliminary design should be of sufficient depth to verify a technical understanding of the problem and to indicate a viable solution. An estimate of the number of pages in each subsection was provided to scope the effort expected of each programmer.

Further, each programmer was required to submit an outline of his subsection. These initial outlines tended to lack organizational structure and consistency, and had to be modified. The chief programmer and integration group reviewed and proofread each programmer's initial documentation. In most cases, the initial documentation required extensive reworking to delete unnecessary detailed processing flow and to provide adequate depth for complex functions.

Previous experience with software design documentation indicated written material tended to lack organization, evaded complex problems, or provided repetitive detail about trivial matters. The documentation outlines and guidelines were an attempt to solve these shortcomings. For future efforts, I would also consider having each major section introduction, summary, and/or conclusion written and reviewed prior to supportive text. This could further scope documentation effort and minimize extensive reworking.

DETAIL DESIGN DOCUMENT

The detail design document of about 1,125 pages took about three months to develop. The preliminary design provided the basis for the detail design which described how the delivered software would be built. The chief programmer developed the overall documentation outline and detail design guidelines. The basic guideline was that the detail design should be a programmer-oriented document that would provide detailed information on how functions were performed. It was to be of sufficient detail to code from, yet general enough to reflect resultant code with a minimum documentation impact. Specific guidelines were identification of all inter-subprogram communication buffers and storage, subprogram overlays, suboverlays, and major routines, subprogram buffers and arrays whose size or structure is dependent on data external to the subprogram, subprogram defaults, and subprogram error conditions and corrective action.

As with the preliminary design, each programmer was required to submit an outline of his subsection. Again, these outlines proved informative but often had to be redone to reflect a hierarchical structure. Each programmer's initial documentation was proofread and returned for reworking. As with the preliminary design effort, I recommend each major section introduction, summary and/or conclusion be written and reviewed prior to the supportive text.

Code and Checkout. Software is frequently evaluated and remembered by its worst attribute (e.g., slow execution, difficult to modify, or awkward to use). Attempts were made to avoid these attributes during the code and checkout phase. Standards, reviews, checkout procedures, and software aids were utilized to enhance development and to produce quality software. The code and checkout phase took about 10 months.

Coding Standards. The basic coding standard was to develop software that was readable, sequential, and maintainable. It is difficult to derive all-encompassing standards due to the numerous peculiar situations and circumstances for which software is developed. Personal judgment is ultimately required to evaluate the detailed software development standards; there is no substitute for good judgment. Mandatory standards were provided regarding routine description, input/output identification, error condition descriptions, and comments. Programming conventions and guidelines were also provided regarding overlay

names, file utilization, variable utilization, variable names, variable assignments, and routine length. The standards, conventions, and guidelines attempted to achieve maintainable code without restricting the experienced programmer's judgment or habits.

Basically, these standards were adhered to; total adherence would require several people full time to monitor each line of code. The effectiveness of these standards will be realized in the future, when program maintenance and modifications are required.

Implementation Reviews. Following successful completion of the customer critical design review, implementation (code and checkout) began. Each programmer's implementation approach and initial coding efforts were reviewed by the chief programmer. Reviews were performed on detailed hierarchy and flow diagrams, and then on listings. These reviews were beneficial, necessary, and a key factor to the success of this software development effort.

It was particularly interesting (and amazing) to review and critique the initial implementation phase. Reviews indicated difficulty in implementing the details of the software design in a manner which was well structured, independent, and easy to check out. That is, the initial software implementation tended to impose additional complexities which could be avoided with a more straightforward approach. Some of the more common faults were:

- Lack of program structure and organization—implementation did not have clear, definable subsets which could be totally verified at intermediate points. Implementation increased program complexity and required extra checkout.
- Misuse of computer resources—one-word disk reads and writes.
- Lack of implementation flexibility—fragile software which was extremely data-dependent.
- Use and misuse of data statements—same data statement used in numerous routines, or buffer sizes not specified in data statements for easy identification and update.
- Redundant code—required extra checkout (vs one subroutine).
- Limited understanding and appreciation for the amount of time it takes to process large volumes of data—data was processed one item at a time.

These are merely personal observations made during the implementation reviews; it is not obvious whether these faults arise from a lack of understanding of the computer and its resources, beginning too close to the task, or not understanding the entire task. Team reviews were not held throughout the development effort; the size and complexity of each programmer's task was such that it would be difficult for other team members to understand and offer constructive detailed criticisms.

Team reviews during the design and especially during the initial implementation phase could alleviate obvious shortcomings and common faults, and could be educational for subsequent development efforts. I'm not convinced team reviews during the entire coding phase are beneficial and economical on all large-scale software development efforts. If code reviews are held, they should be performed by a code review board (vs entire team); the board approach appears to be necessary if programming requirements impose mandatory coding standards.

TEAM CODE CRITIQUE Upon completion of the software development effort, each programmer was given representative listings of code generated by all other programmers on the team. These listings were reviewed and critiqued in a team review meeting. The intent of this review was to expose the development programmers to software maintenance and enhance their judgment of coding techniques.

The codes received a variety of comments. Some were described as well structured, maintainable, and easy to follow, with a few singled out as easier to read and follow due to the linearity of the task. A number received descriptive processing flow comments. Programmers pointed out restrictions or error conditions, occasional lack or misuse of data statements and external storage resources, and indicated where programming notes should have been more frequent and descriptive.

The effectiveness of these reviews will be realized in the future, when team members develop new software.

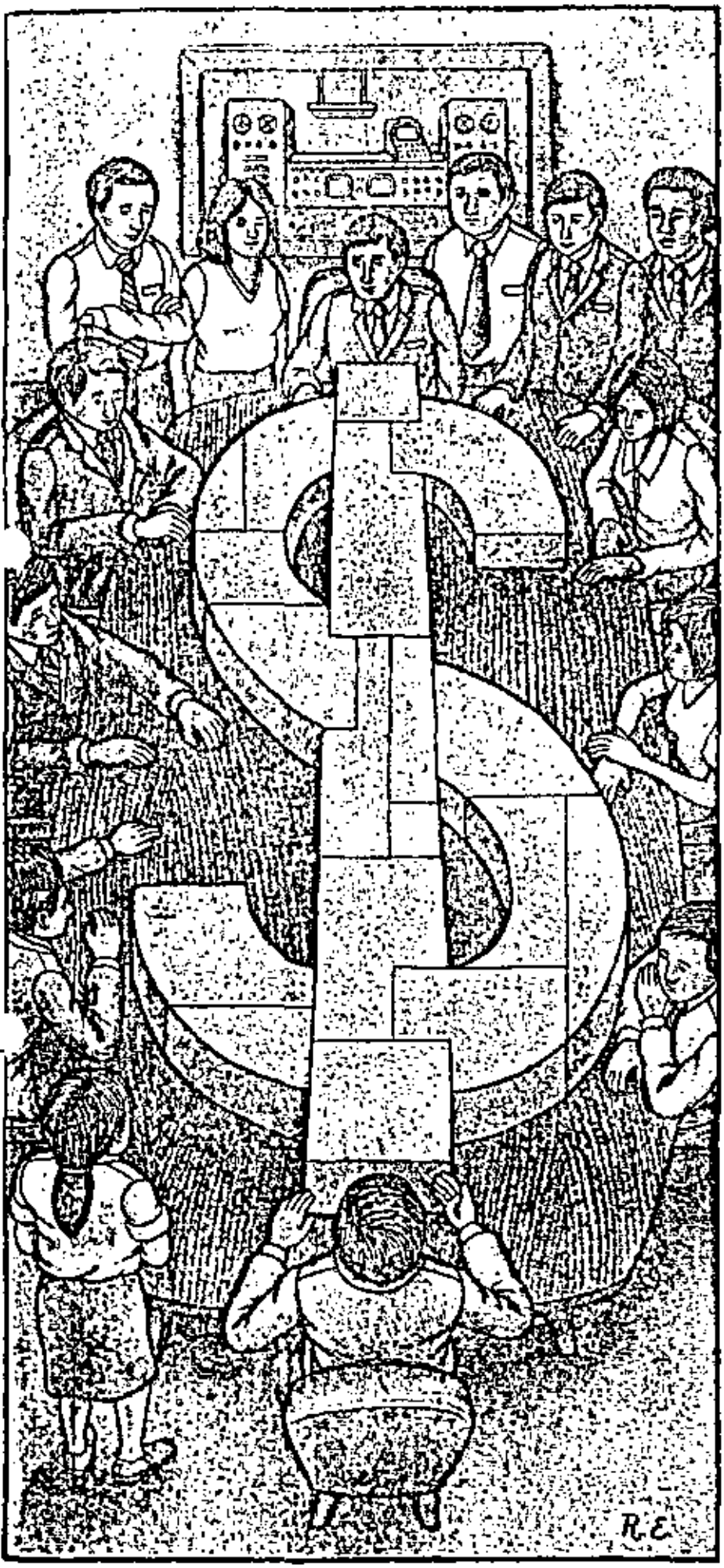
Top-Down Coding/Checkout. Code and checkout progressed in a top-down manner. Initially, the program executive was coded and totally checked out; "stubs" for all overlays were incorporated, and utility/library routines completed. Main overlays and suboverlay "stubs" were coded and checked out by the responsible programmer. Subsequent code and checkout sequences were left to the discretion of the individual programmer; emphasis was placed on completing the more complex and interoverlay/suboverlay functions first.

Software Aids. A useful set of software development aids and utility routines were factors in the overall success of the programs. The following are noteworthy:

- Display processing simulation program. This batch program saved significant checkout time by enabling the programmer to simulate execution of display subprograms without actually operating the display.
- Display program central memory, external storage, and internal file dump features. These online features provided the capability to dump pertinent data if errors were encountered during display checkout without subsequent diagnostic trace updates.
- Library containing frequently utilized functions such as data bit manipulations and disk access routines.
- General purpose file program used to create checkout files.
- Central file book. This book contained all pertinent file formats and specifications; modification to files were coordinated through one individual.
- Interactive computer terminals. Terminals significantly improved daily computer turnaround.

Testing. The system engineering group was responsible for the final formal program testing and demonstration. This test was deficient in that it did not demonstrate all the program's capabilities, or a realistic operational scenario; however, the test did identify several program discrepancies. The number of program discrepancies did not appear to be significantly less than on previous development efforts. Strict quantitative comparison as to the number of discrepancies is not the single most important software development evaluation criterion. Evaluation criteria should realistically include such things as requirement satisfaction, design flexibility, code maintainability, and fault isolation simplicity.

Software development visibility and realistic operational applications could be better demonstrated by distributing formal testing throughout the entire development phase. These incremental tests could include combinations of top-down, diagnostic, and subprogram component tests. (Diagnostic tests exercise program limits, error messages, conflicts, input anomalies, and abnormal operational sequences.) The formality of these incremental tests must be traded off with development costs and schedules. Effective software testing requires an understanding of the requirements, operations, and software design.



SOFTWARE CONTROL TECHNIQUES

Software control is a topic of much concern, especially on the development of large-scale computer programs. Effective software control requires a very detailed understanding of the effort, a design which may be partitioned into separate independent subprograms, and frequent periodic monitoring. Theoretical control techniques provide basic software management guidelines; the identification of applicable control techniques requires a range of programming and implementation experience.

This experience provides the insight required to develop effective software control techniques which fit the needs of the development effort. A single control technique is not adequate to effectively monitor and control a large-scale software development effort. Effective and realistic software development statusing and control require a combination of several independent techniques and objective evaluations. The following techniques were used during the code and checkout phase of this effort.

Completion Date Estimates. The hierarchy diagrams were used to identify tasks to be estimated. The responsible programmer for each task was requested to make an estimate as to the number of instructions and how long it would take to code and check out the task. Estimates were required for each task, subtask, and subroutine. Weight factors were also estimated to provide a subfunction's relative magnitude and complexity. The intent of having the responsible programmer estimate the effort was to force a more detailed analysis of the task and, hopefully, to instill a personal commitment to the estimate.

With allowance for the fact that programmers tend to be optimistic, the estimates were reviewed and a work schedule generated (see Fig. 2). Each week, the responsible programmer estimated the percent complete on each subtask. The chief programmer evaluated the reported percentages and estimated overall status of the task; the number of weeks ahead or behind schedule was recorded.

Checkout Milestones. Intermediate checkout milestones were established for each task. These consisted of 10 to 15 specific items or functions which could be verified with printer outputs, dumps, or on the displays (e.g., subroutine "A" complete, subtask "1" file generation complete, task "A" vector generation subfunction complete). These milestone dates were relatively evenly spaced throughout the code/checkout phase, and were consistent with the completion date

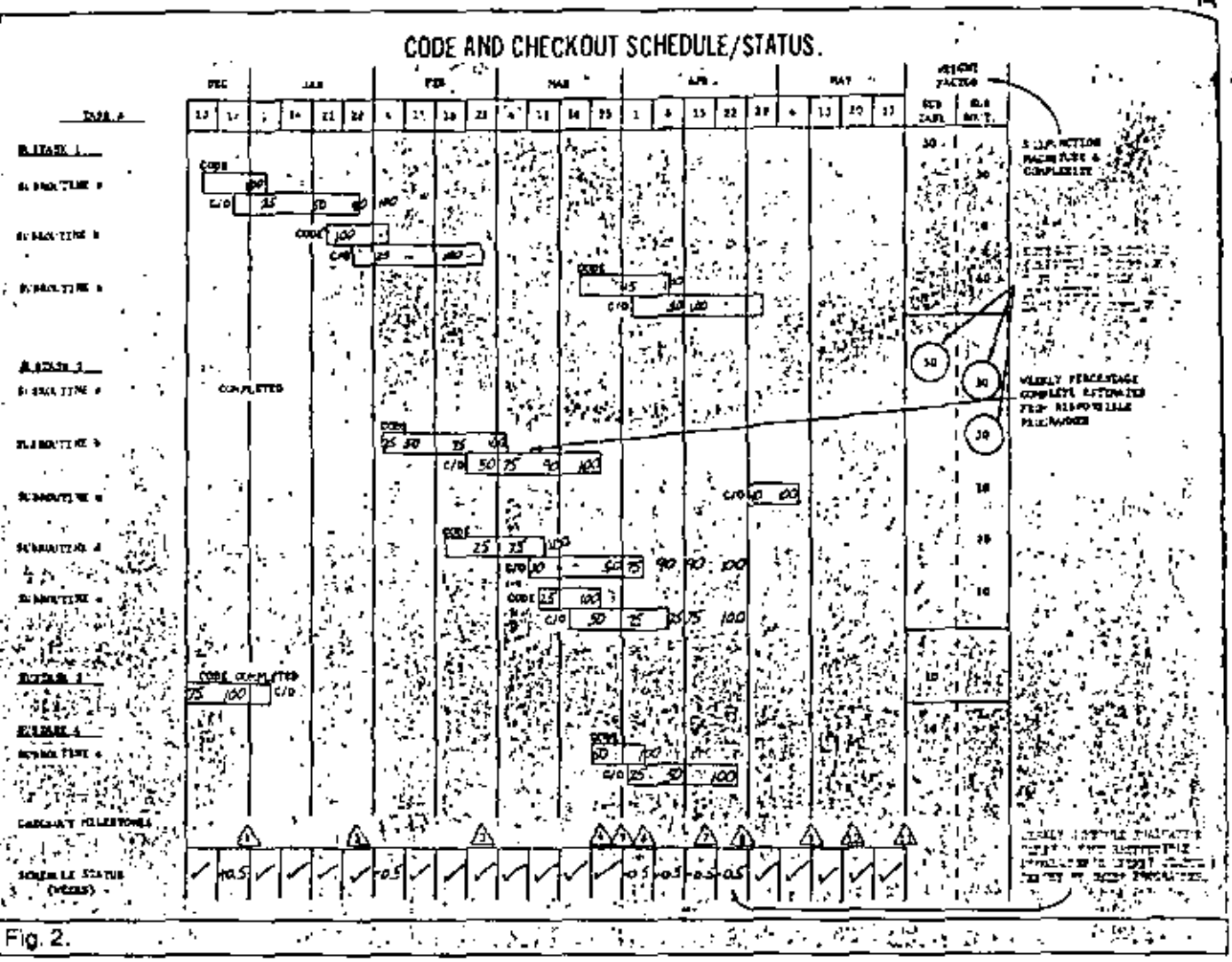


Fig. 2.

estimates. The objective of these milestones was to impose smooth productivity throughout the development phase, and to more clearly indicate schedule problem areas. These milestones were defined such that the item was a GO or NO GO situation; this alleviated subjective evaluation of the task as percentages complete.

Time-Line Charts. Time-line charts were used to indicate dates when "stubs," functions, and subfunctions were to be incorporated. These charts were simply bubble charts with associated dates and estimated critical paths. The dates on the time-line charts were consistent with the completion date estimates and checkout milestones.

Weekly Status Reports. Weekly status reports were required from each programmer; these contained an estimate of the code and checkout completed, status of checkout milestones, and written text regarding progress and problems encountered during the week.

These weekly reports helped to isolate problems and slippages so corrective action could be taken. Theoretically, the programmer updates the "estimated completion dates" as the programming effort progresses; this doesn't work too well with an overall fixed completion date. All schedules were developed with about a one-month margin; tasks which began to slip were monitored more closely. Minor slips (one to two weeks) were tolerable, but positive attempts were made to assist the programmer through job prioritization (keypunch, and computer turnaround), data analysis assistance (software integration personnel), and in a very few cases, overtime was allowed (less than 0.01%).

The programmers seemed to respond to this active concern for their tasks. Major slips required task partitioning and assignment of subtasks to other programmers; the design allowed this partitioning with minimum overall impact. Two major slippages were noted early in the development schedule, when task reassignments could easily be implemented.

A discrepancy log identified problems encountered during checkout by the engineering and integration groups. The intent of this log was to keep track of discrepancies and assure that they were being solved and not "forgotten." This log contained the date the discrepancies were encountered, description, and responsible programmer(s). The weekly number of open discrepancies per programmer seemed to vary significantly (from two to 25); the number appeared to be inversely related to the individual programmer's expertise.

MAKING COUNT ESTIMATES

Historically, accurate instruction count estimation has been a difficult task on large-scale software development efforts. Estimates and actual final instruction counts tend to differ by orders of magnitude. This instruction count difference is probably the area of greatest variance in large-scale software development efforts, that is, the area most likely to be wrong by the largest magnitude. The procedure for estimating instruction counts must be more accurate *if future software development efforts are to be realistically costed/priced according to these estimates.*

The estimation procedure is further complicated since a given task can be coded by different programmers and may result in an order-of-magnitude difference in the final instruction count. For this software effort, an "instruction" is defined as an executable or declarative source statement (comment or continuation cards are not "instructions").

Monthly Instruction Productivity. Source instruction count totals were obtained for each programmer and task on a monthly basis. These counts gave some indication as to how the overall effort was progressing and the productivity of the individual programmers.

During the peak coding months,

individual programmer productivity ranged from 150 to 2,000 instructions a month, with a nominal range of 400 to 1,000. The average productivity for the code and checkout phase (10 months) was approximately 490 instructions a month.

"Total cards" (instructions and comments) is occasionally a useful productivity figure. The "total card" average productivity for the code and checkout phase was approximately 890 cards a month. Programmer productivity varied from 680 to 1,360 cards a month.

Instruction Estimation Technique. Originally, each task was separated into subtasks and functions (as derived from the hierarchical diagrams). These functions were then evaluated against comparable functions in existing programs for which actual instruction counts were available. Estimates were made for program overhead and for functions which had no comparisons. These counts were summed to determine original program instruction estimates, but the estimates were as much as 100% off. Throughout the coding phase, the instruction count estimate was updated, based on actual monthly instruction counts and programmer estimates of percent code completed (see Figs. 3 and 4).

The display program code was reported 100% complete during the eighth month, yet the actual instruction count increased about 10% over the next two months.

The structured concepts, control techniques, and personnel led directly to the success of this software development effort. The top-down design and hierarchy diagrams provided the foundation from which the software was developed, managed, and controlled. A more structured top-down approach to feasibility-study report documentation and testing could enhance all aspects of a software development effort. The chief programmer and team concept was successful because of each individual's ability and willingness to meet the demands of the task. Documentation guidelines and reviews enhanced the quality and usefulness of the software documents. Additional guideline refinements and intermediate reviews appear necessary to minimize extensive documentation rework.

In-depth team reviews on large systems do not appear to be totally effective due to size-comprehension limitation; constructive detailed criticism appear inversely related to program size. The status and control techniques offered credence to the development progress and imposed steady productivity. These techniques provided the visibility and confidence required to manage the development effort and to achieve an on-time product delivery.

Don't expect immediate success and miracles because structured techniques and concepts are used on a software development effort. The value of structured techniques is to impose some order and discipline; ultimately, it is the individuals who govern the final outcome. Don't implement the concepts and techniques for the sake of being "structured"; one must recognize their utility and applicability before they can be successfully implemented. Software development and control techniques are evolving. The concepts and techniques incorporated in this effort fit its specific personalities and needs. ◻

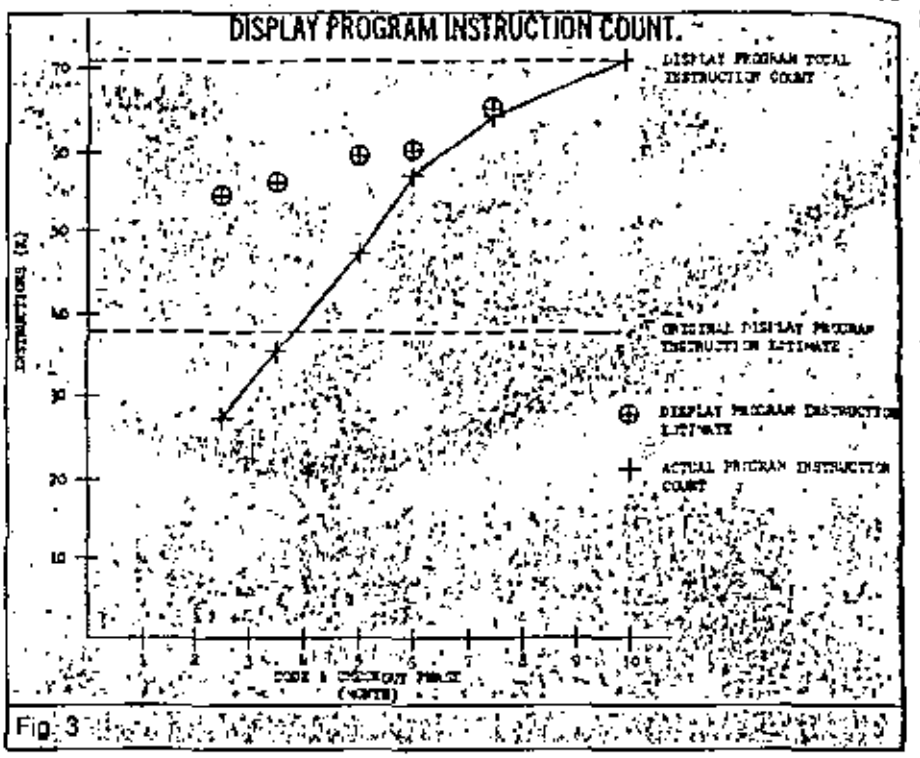


Fig. 3

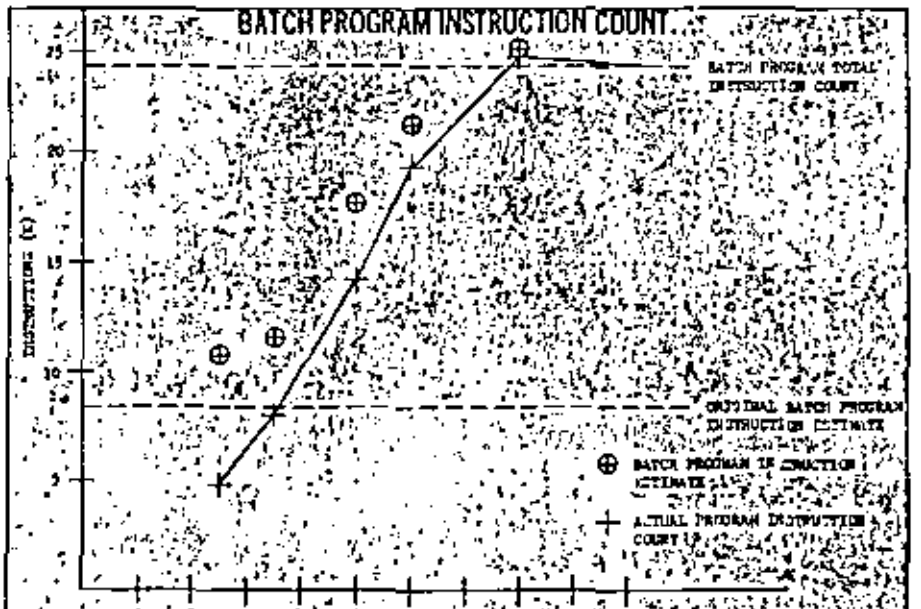


Fig. 4

DAVID S. IWAHASHI

Mr. Iwahashi is a staff engineer in software design and development at Lockheed Missiles and Space Co., Inc., in Sunnyvale, Calif. Since joining LMSC in 1966, his projects have included systems analysis and design, developing and maintaining programming, and software standards and requirements.

Structured Systems Analysis: A Technique to Define Business Requirements

Kathleen S. Mendes Exxon Corporation

Structured Systems Analysis (SSA) is one of an integrated set of techniques that has resulted from Exxon's research in software methods. The author discusses SSA's role in system development, describes its graphical language, and explains how users and computer analysts use it to model business operations and determine requirements for computer applications. She then describes diverse business and technical projects that illustrate SSA's efficacy. The author suggests that SSA — originally developed for use in the initial stages of system development — has become a general-purpose modeling technique, which has helped to further establish formal methods for system development. Ed.

Growth in the development and use of computer systems has been rapid over the past twenty years. Hardware costs, for a given level of performance, have decreased. Communications technology, an important factor in the advance of computers and information processing, has expanded the capabilities for sharing and coordinating information from many sources. However, software technology — the methods and languages used to develop computer systems — has not shown comparable improvement. As a result, the hardware revolution has driven the demand for computer systems upward by providing increasingly attractive opportunities, but software capability has constrained the ability of system developers to meet this demand.

These hardware and software trends motivated Exxon's computer scientists, in the mid-1970s, to try to improve software methods. Their objective was to devise systems analysis and design techniques that would enhance both the quality of computer applications and the ability of development staffs to deliver systems.

Structured Systems Analysis (SSA) is a member of an integrated set of techniques that resulted from Exxon's research and development in software methods. SSA is a business modeling and communication technique used jointly by users and computer systems analysts to describe business environments and to formulate requirements for computer applications. It gives analysts, for the first time, both a definition of the products to be generated at each stage of a project, and a basis for project planning. Moreover, its application extends beyond the system development environment. It has been used effectively in business improvement studies that do not involve computerized solutions.

This article describes Exxon's SSA, illustrates its usage, and discusses its benefits. It also suggests that SSA, originally developed as an analysis technique for use during the first stages of system building, has evolved into a general purpose modeling technique for describing a business.

Background and Development

In the past, due to the absence of formal analysis and design techniques, analysts learned through apprenticeship. System developers worked with more experienced people until they developed their own approaches. The first formalization of the system building process introduced the project management disciplines of other engineering and technical areas into computer application projects. These disciplines regarded system development as taking place in four general stages: *Analysis*, in which a business is described and the requirements for a computer system are formulated; *Design*, in which the hardware and software specified in Analysis are configured; *Construction*, in which the software modules are implemented and the hardware is installed; and *Maintenance*, which encompasses a broad range of activities related to keeping the system operational. However, they failed to provide a methodology that could accomplish the objectives of each of the stages of system development.

During the late 1960s, software engineering pioneers, such as Edsger Dijkstra and Michael Jackson, began fundamental research on formal approaches for system development. Much of the research focused on program design and implementation, key ac-

Kathleen S. Mendes is Senior Project Analyst in the Communications and Computer Sciences Department of the Exxon Corporation. Ms. Mendes holds the B.A. degree in mathematics from Manhattanville College and the M.B.A. degree from the Graduate School of Business of Rutgers University. In her present position, she has participated in the research, development, and instruction of Exxon's Structured Systems Analysis Technique, and has contributed to the establishment of a long-term strategic plan for software technology. Ms. Mendes has been a part-time faculty member of the Management Sciences Department of Kean College, and has published numerous Exxon proprietary documents.

tivities in the Construction stage of application building. In the mid-1970s, Exxon began to apply the results of this work to its own environment. Its main purpose was to establish an integrated set of system development techniques. The initial effort led to the introduction of PST (Program Structure Technology) to Exxon's worldwide community of system developers. PST, adapted from Michael Jackson's "Program Design Technique," could provide a precise description of a program and effectively communicate the program's design and behavior.[1] Its graphical notation was a key factor in the widespread usage of PST, and it became a well-established standard within two years.

In the mid-1970s, empirical evidence began to confirm the important role Analysis plays in the building of high quality computer applications. The results of one study indicated that more errors are introduced into a system as a result of failures in Analysis than as a result of failures in any other system building phase.[2] The conclusion of another study was that Analysis errors are more costly to correct and have a greater impact on the effectiveness of the system than errors introduced during either Design or Construction. Significantly, the effects of Analysis errors do not cease with the implementation of the system, but carry over to Maintenance.[3]

Therefore, any methods that improve the quality of Analysis increase the effectiveness of the system development process. Such methods, by contributing to more efficient use of time and by reducing the need to gather additional information as the project progresses, also increase the productivity of systems analysts — usually the most highly paid members of the data processing staff. In the late 1970s, Exxon's computer scientists focused their efforts on developing an Analysis methodology. The purpose of this methodology was to provide a disciplined approach for Analysis, as PST had done for program design and implementation.

Concurrently, application analysts, who were using PST, began to experiment with the flexibility of the notation. They began to use the PST diagramming style during the Analysis stage of system development to specify requirements. Based on the success of this experimental work, Exxon's computer scientists decided to use this approach to formalize system requirements analysis. Thus, the PST notation was brought forward from Construction to Analysis, and it was generalized to provide the basis for the present SSA technique.

Having established a commonality of approach and notation, SSA development proceeded in an evolutionary manner. Existing theory and concepts were drawn upon as needed. They were then adapted and integrated into the technique. At major checkpoints in its development, SSA was tested on pilot projects. This testing contributed significantly to the quality and usability of the present technique.

SSA and Other Requirements Analysis Techniques

The development of a number of requirements analysis and specification techniques during the past several years indicates the importance of these techniques to the data processing industry. Some of the more well-known specification approaches are Softech's SADT ("Structured Analysis and Design Technique"); Yourdon's "Structured Analysis Technique"; Sarson and Gane's "Structured Systems Analysis"; and the University of Michigan's ISDOS ("Information System Development and Optimization System").[4]

The fundamental difference between SSA and these methods is one of orientation. SSA approaches Analysis from the viewpoint of the business. The other techniques approach it from the perspective of data processing. SSA can represent completely in a clear, concise model all aspects of the business operation. Other methods are incapable of providing such complete business models.

Another significant difference between SSA and other structured techniques is the graphical notation. The SSA graphical approach consists of several diagram types, while the techniques of Softech, Yourdon, and Sarson and Gane rely upon a single style

of diagram. Analysts using SSA have indicated the advantages of its diagrammatic approach. It gives more insight into the business by providing different views, each of which contributes to an integrated business model. Each business entity is described by a notation that is appropriate for that entity's level of decomposition and for its role in the model analysis to follow.

SSA: A Description

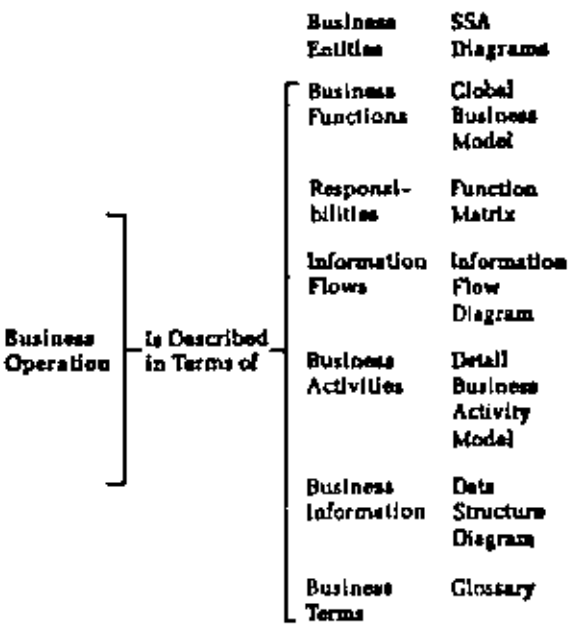
SSA is an analysis technique used to understand the business operations of planned computer applications. Its purposes are to model, document, and communicate to users the requirements of computer applications, and to specify the capabilities to be delivered by the recommended system. It consists of a *graphical language* — a precise notation and set of diagrams which collectively are used to model a business — and a process — a step-by-step method for building and verifying the model. These components provide a formal, self-documenting approach to the Analysis stage of system development. The graphical language supports information gathering and analysis. The process assists in modeling the operation as it currently exists. The model is then modified to include the new functions provided by the computer application. The revised model and the supporting text replace the traditional specification document.

SSA has been used at Exxon for two years. During this time it has been effective in a wide range of applications that have varied in size, complexity, and orientation. It has been scaled to suit the needs of individual projects, and has been used effectively in a diversity of environments — including both business information systems and technical computing applications.

SSA's Graphical Language

The SSA graphical language provides a versatile vocabulary for model building and analysis. As shown in Figure 1, the style of

Figure 1 SSA Graphical Language



the model consists of several diagram types: hierarchies, matrices, and network flows. Figure 1 also illustrates the relationship between the SSA model diagrams and the business entities they describe.

A business model can be represented with the following diagrams and definitions:

— *A Global Model* to describe the overall *logical relationships* among the business *functions* in the organization under study.

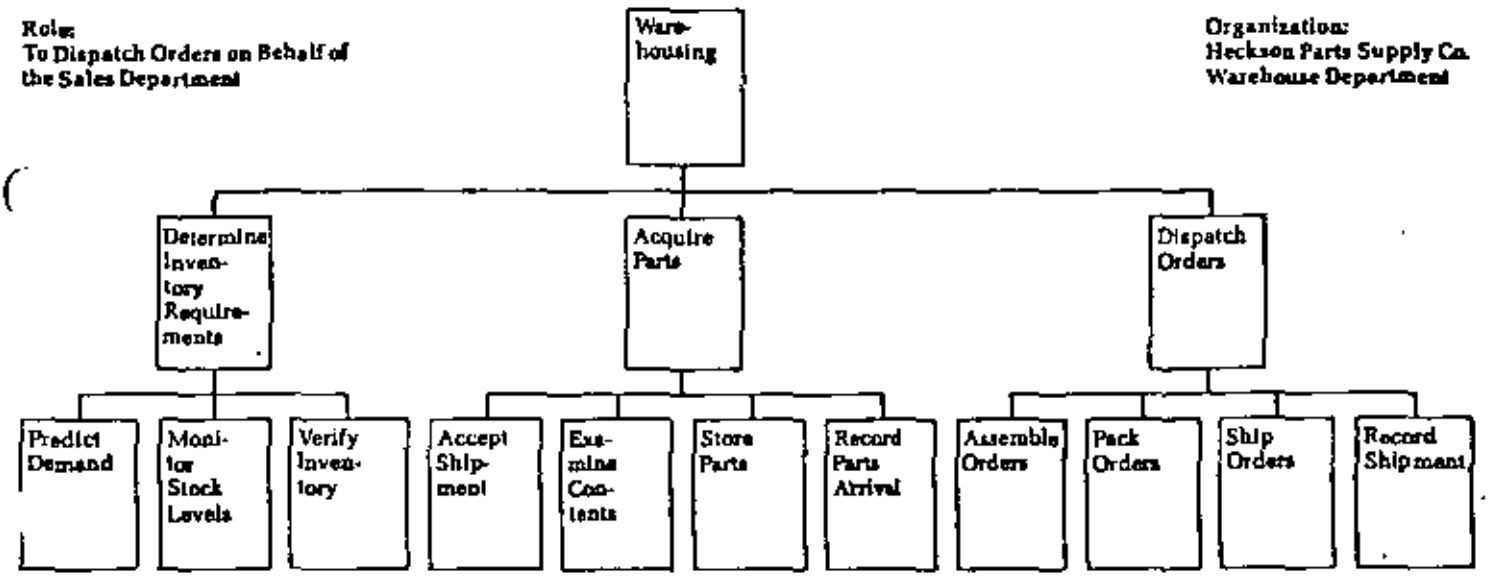
— *A Function Matrix* to define the *responsibilities* for each function identified in the Global Model.

— *An Information Flow Diagram* to represent the *flow* of business information.

— *A Detail Activity Model* to describe how activities, which correspond to the lowest-level functions in the Global Model, are carried out.

— *A Data Structure Diagram* to describe the logical structure of business information as viewed by the user.

Figure 2 Global Business Model



—*A Glossary of Business Terms* to define terminology common to the business operation.

The remainder of this section describes SSA's graphical language. Subsequent sections explain how to build model diagrams and how to use them to identify and specify solutions. Each diagram is discussed in the recommended sequence of construction.

Global Model. The Global Model of a business is a functional decomposition (a breakdown of *business functions* by purpose) described in a hierarchy of three to five levels. (Figure 2 shows a Global Model of a warehousing business.) In SSA, a *business* is defined as a logical set of functions which exists to provide a product or service (e.g., Warehousing). A *function* is a group of logically related activities (decisions or tasks) required to manage the resources of the business (e.g., Assemble Orders). In addition, a Global Model contains a single-sentence "role statement" capturing the nature of the product or service supplied by the business, the client or market it supports, and its economic commitment. The Global Model concept was developed at Exxon by integrating Constantine and Yourdon's and Myers's functional decomposition techniques with concepts derived from IBM's "Business Systems Planning Methodology."[15]

Function Matrix. A Function Matrix maps the logical organization of the Global Model to the physical organization chart. (Figure 3 shows a sample Function Matrix which corresponds to the Warehousing Global Model in Figure 2.) All of the business functions are listed down the left side of the figure. Across the top are the *responsibilities*, i.e., the job descriptions, organizational units, or personnel accountable for the performance of the function. An "X" at the intersection indicates which functions are performed by each group.

Figure 3 Function Matrix
Warehousing

Function \ Responsibility	Warehouse Manager	Receiving Clerk	Forklift Operator	Shipping Clerk	Picker	Stock Clerk
Warehousing	X					
Determine Inventory Requirements	X					
Predict Demand						X
Monitor Stock Levels						X
Verify Inventory						X
Acquire Parts		X				
Accept Shipment			X			
Examine Contents		X				
tore Parts			X			
..ecord Parts Arrival						X
Dispatch Orders				X		
Assemble Orders					X	
Pack Orders				X		
Ship Orders			X			
Record Shipment						X

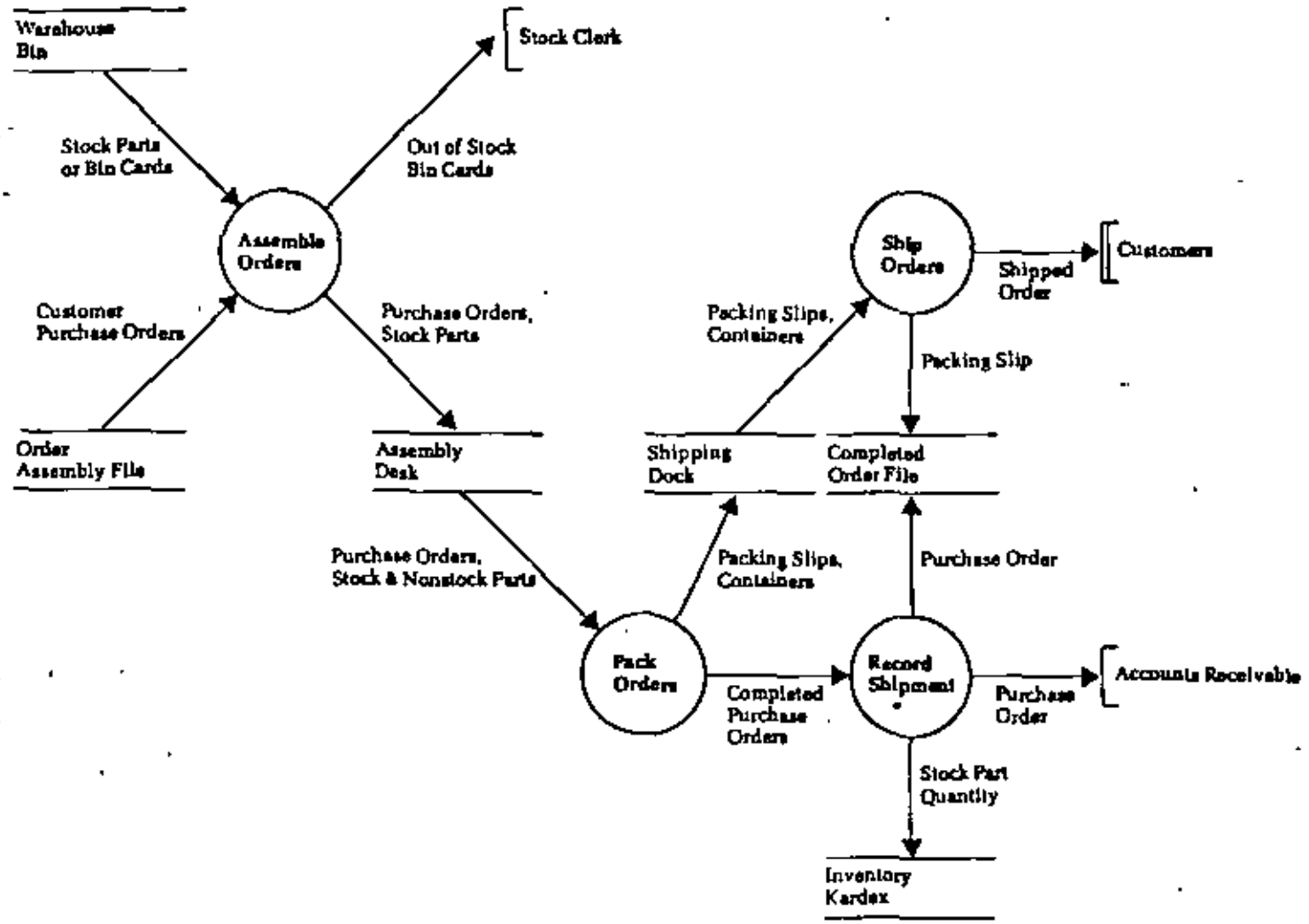
Information Flow Diagram. A Flow Diagram is a network representation showing the flow of business information. A flow is the transfer of information or material between business functions. The sources or destinations of these flows can exist within or outside of the business. At a minimum, one Flow Diagram is drawn for the lowest-level functions of each Global Model subtree. Figure 4 is an example of an Information Flow Diagram for the Dispatch Orders subtree of the Warehousing Global Model. It describes where information is used, what it is used for, and how the functions interact. Flow Diagrams can be annotated with statistics on volumes, resource usage, and critical timing. Optionally, Flow Diagrams may be drawn to describe the relationships among the parent functions of the subtrees. These relationships are referred to as high-level flows.

The notation used in Flow Diagrams represents information or material flows; business functions; information stores (passive repositories of information or material which can be either automated or manual); sources or destinations of information or material outside of the business operation; and sources or destinations of information or material outside of the network but inside of the business operation. The basic format of Flow Diagrams derives from the "Data Flow Diagram" concept developed by Yourdon and DeMarco.[6] However, their usage and notation are modified in SSA.

Detail Activity Model. A Detail Model specifies the relationships between activities and conditions under which activities are performed. An activity is a well-defined unit of work — a task or a decision (e.g., notify stock clerk if part is out of stock). A Detail Model is drawn for each of the lowest-level functions on the Global Model. Figure 5 is an example of a Detail Model for the function, Assemble Orders. It is a hierarchical breakdown by function, which is annotated to describe relationships among activities. Its symbols represent such relationships or conditions as hierarchic membership (with an implied or unspecified sequence); repetition; exclusive alternatives; inclusive alternatives; explicit sequence; parallel activities; release or critical timing condition; and termination activity. The original concept of the annotated hierarchy was derived from Michael Jackson's Program Design Technique.[7] Exxon extended Jackson's notation to the Detail Model's description of business processes.

Data Structure Diagram. A Data Structure Diagram is prepared for each unit of business information that is identified on the Flow Diagrams. *Business information*, as defined in SSA, includes organized data aggregates such as documents, files, products, and intangible forms of information (e.g., order

Figure 4 Information Flow Diagram
Dispatch Orders



Key:

→ — Information or material flows

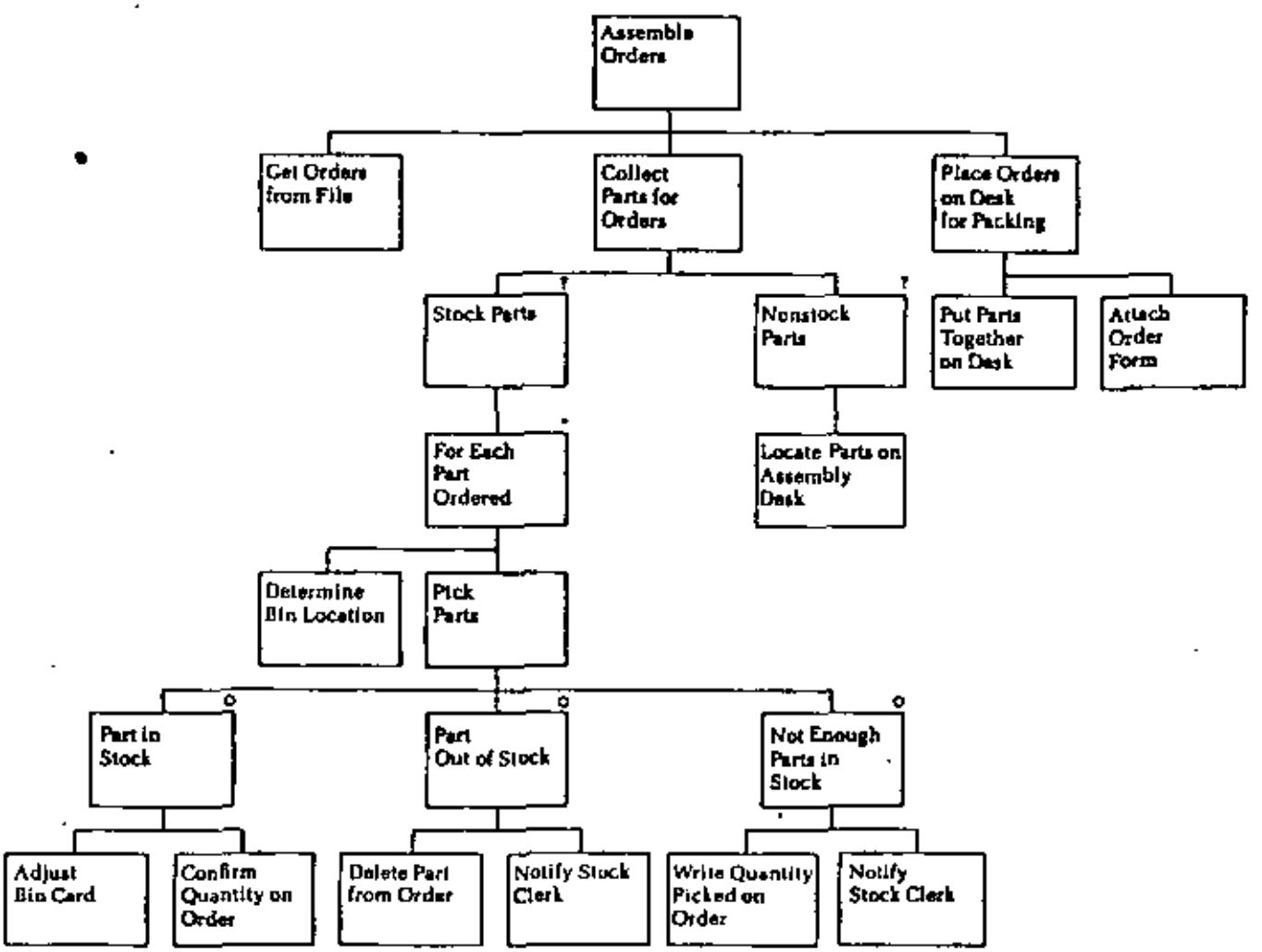
○ — Business functions

— Information stores

] — Sources or destinations of information or material outside of the business operation

] — Sources or destinations of information outside of the network but inside the business operation

Figure 8 **Detail Business Activity Model**



Key:

blank — Hierarchic membership with an implied or unspecified sequence

* — Repetition

o — Exclusive alternatives

? — Inclusive alternatives

> — Explicit sequence

— Parallel activities

— Release or critical timing condition

↓ — Termination activity

forms, credit criteria). Figure 6 shows an example of a Data Structure Diagram of an order form as viewed by the clerk who takes orders. It is used to gain a deeper understanding of the information required to support the tasks and decisions involved in operating the business. A Data Structure Diagram is a hierarchical data decomposition, which may be annotated with statistics on data volumes and frequencies. It describes the "parent-child" relationship of the data by indicating hierarchic membership, repeated occurrence, mutually exclusive alternatives, and inclusive alternatives. It also includes a symbol for a dimension operator which is used in special matrixlike cases. The concept and notation originated from Michael Jackson's Program Design Technique. As used in SSA, it has been generalized and expanded.

Glossary of Business Terms A Glossary is compiled throughout the model building process. It is used to eliminate multiple definitions and ambiguities, and it serves as a basis for discussion and clear communication between users and analysts.

The Process of SSA

The SSA process is a step-by-step approach that guides the analyst and user through systems analysis. It has both the flexibility needed to develop a structure of business operations and the formalism needed to build a model. It also provides criteria for evaluating the model's correctness.

Figure 7 is a schematic representation of the SSA process. It shows the sequence of building the model diagrams. This sequence uses a top-down approach that starts with the most general (the Global Business Model) and proceeds to the most detailed (the Data Model). The figure also shows the recycling loops needed to refine the diagrammatic representations. During information gathering, the business model is the medium of communication between the user and analyst. The model is completed when it is confirmed that it fits the users' logical views of the business. Once verified, the model becomes the basis for identifying problems and formulating solutions.

SSA moves business and systems analysis from an intuitive to a teachable, structured, and concrete approach. This advance is the result of a well-defined process that assists the analyst in *building, verifying,* and *analyzing* a business model.

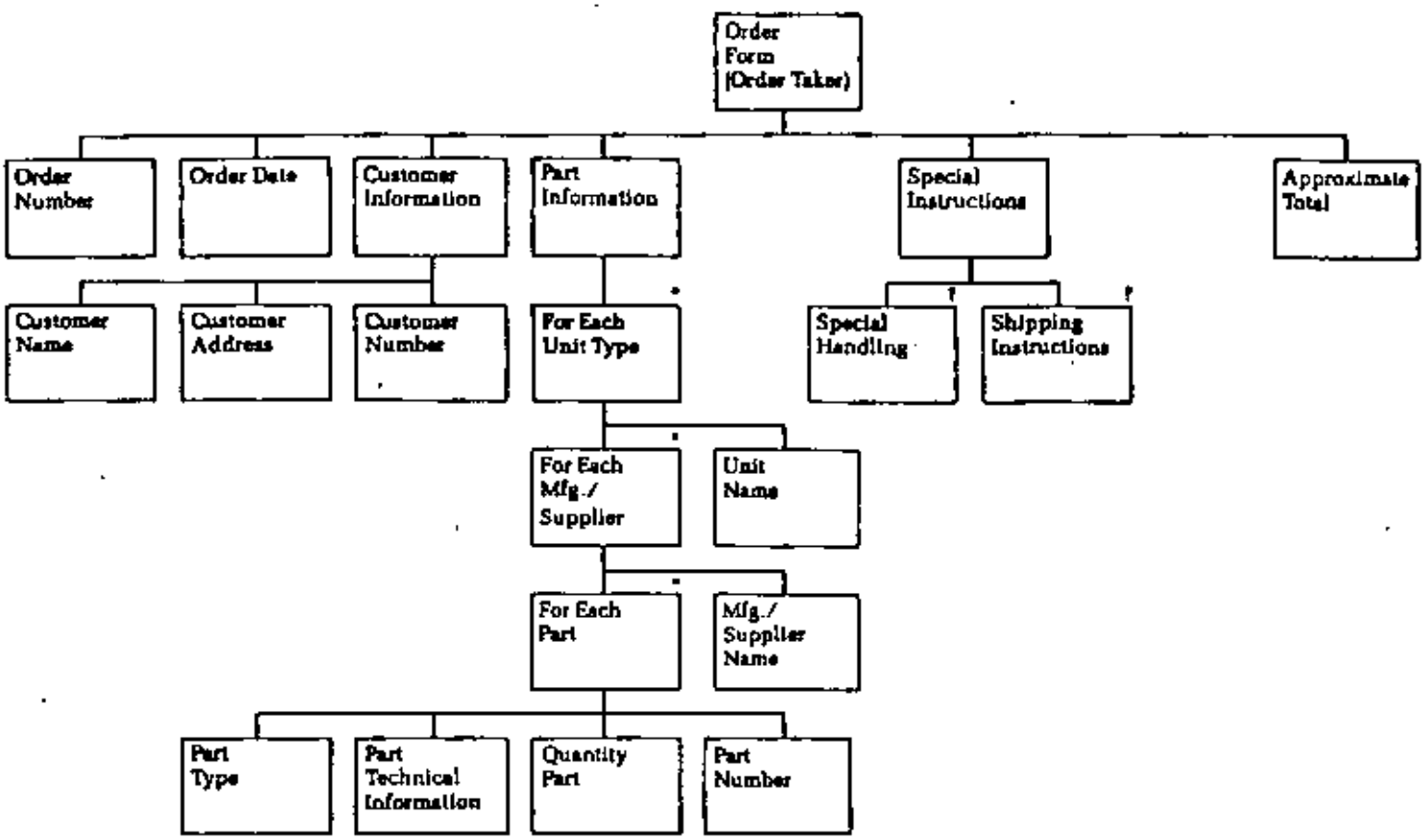
Model Building. Building an SSA business model is a cooperative effort between analyst and user. The diagrams are constructed during fact-gathering interviews, and serve to describe the current business in the project reports. The analyst uses SSA guidelines to elicit the relevant information, record key features, and eliminate irrelevant detail. The interviewing structure fosters consistency in communications between analysts and users. The diagram building approach establishes the set of information to be gathered and thereby provides a framework for each interview. Preestablished questions concerning business objectives, problems, future plans, and business environment forecasts can be drawn upon as needed.

SSA provides procedures to determine the sequence for building diagrams. Model diagrams generally are constructed in the order indicated in Figure 7. However, flexibility exists at the lower levels. For example, in a data-driven business, the Data Model is built before the Detail Model. Nevertheless, the relationships between diagrams always remain intact. Depending upon the size and objectives of the project, SSA can be used to produce a scaled description of the business. This description includes, at minimum, the Global Model and Information Flow Diagrams.

The rules and syntax for using the SSA notation are the grammatical foundation of the SSA graphical language. For instance, these rules provide that special symbols may not appear on the Global Model; a role statement must be a simple sentence; functions and activities must be described in verb-object form; and error conditions should be excluded from Flow Diagrams.

SSA also provides guidelines for decom-

Figure 6 **Data Structure Diagram**



Key:

blank — Hierarchic membership

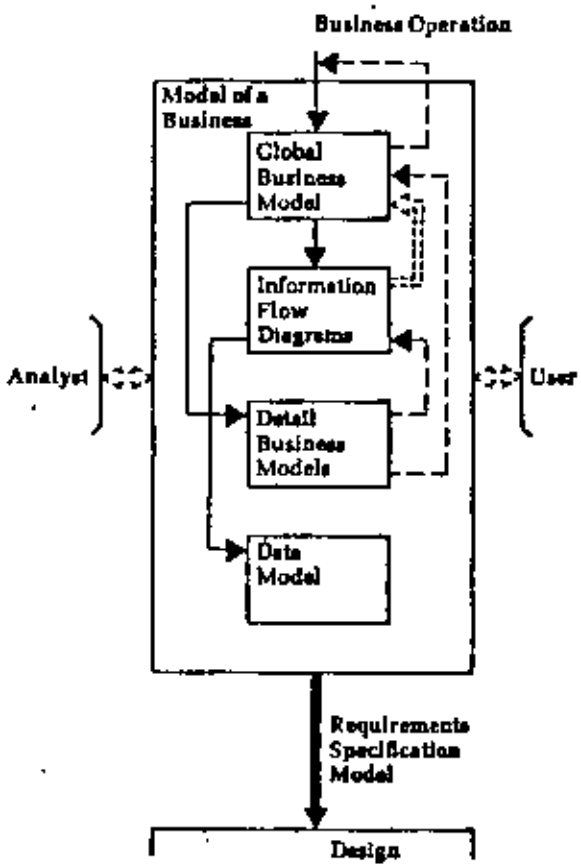
* — Repeated occurrence

o — Mutually exclusive alternatives

? — Inclusive alternatives

& — Dimension operator — used in special matrix-like cases

Figure 7 SSA Process



posing functions and activities. For example, the following rules determine the functions on the first level of the Global Model hierarchy:

— The first function must cover the requirements aspects of the business; that is, it must show how resource planning and acquisition are carried out (e.g., Determine Inventory Requirements in Figure 2).

— At least one supporting function must be identified that describes the business's provisions for maintaining its service or product (e.g., Acquire Parts in Figure 2).

— The last function must be a disposition function which describes the business's termination of responsibility for a product or service (e.g., Dispatch Orders in Figure 2).

Similar rules apply in the breakdown of the Detail Model. Without criteria such as the above, functional decomposition can be difficult, requiring subjective judgments and experience.

Model Verification. Building the model diagrams is an iterative process. As such, criteria are applied to the model to determine if and how the model can be improved. Once the criteria have been satisfied, the model is ready for confirmation by all users. ~~Closure is established when all levels of users have verified it.~~

Criteria to determine whether the model has been decomposed to the appropriate level are applied to the model diagrams. For example, Flow Diagrams that have multipath connections (two functions that pass information back and forth) suggest that further decomposition of the Global Model is required. Or, if a multipurpose function (one with multiple flows in or out and having more than one information transformation) can be identified, then the model requires a further breakdown.

Criteria are also applied in order to test for the correctness of the graphical representation. An analyst can evaluate the model diagram, for instance, by seeing whether or not each function is labeled descriptively. The function should not be defined in terms of its output flow, as described in the Flow Diagram, but should be named to reflect its purpose. The analyst also can ask: Have laws of completeness and logical consistency been met? For example, does every information store have an input and output? Are the specified inputs of a function sufficient to produce the indicated output? Is all flow-related information appropriately represented on both Flow Diagrams and Detail Models?

Adherence to the SSA model building and verification process yields a technically cor-

rect, self-documenting description of the business operation. Moreover, an analyst's use of SSA leads to models that are the equivalent of those produced by others who use the technique.

Model Analysis. After the SSA model of the business has been verified by users, it becomes the basis for *diagnosing business problems, identifying computerization opportunities,* and *specifying the requirements model.*

Diagnosing Business Problems
The first step of this analysis addresses those aspects of the business that have been cited as problems by users. In proceeding, it is important to evaluate the organizational and control structure of the business. This analysis can be made by comparing the Global Model and Function Matrix with the organization chart. The comparison shows if the business is organized functionally, if responsibility and control are clearly allocated for each function, and if the geographic distribution of the business makes sense in terms of functional interaction.

It is also important to estimate work loads and peak loads. One can apply simple formulas, using the statistics collected on the Flow Diagrams, to determine if functions are under or overworked; to uncover the source of backlogs; or to trace the cause of high error rates. The Detail Models can be used to find ways to reduce demands on people. Eliminating unnecessary tasks or transferring certain tasks to the computer, for instance, could accomplish this reduction. The Flow Diagrams and Data Structure Diagrams can indicate ways to increase the capacity of staff, such as improving documents, forms, and human-computer interfaces.

Finally, it is important to establish data requirements. The model diagrams can be used to determine the content and structure of the information required for each business function. By contrasting the data actually used by a function (as represented in the Detail Model) with that passed to it (as described in Flow Diagrams and Data Structure Diagrams), one can identify redundant, inconsistent, incomplete, and improperly structured data.

Identifying Computerization Opportunities
It is often necessary, in solving business problems, to determine where the use of computers is feasible and cost-effective. In making this evaluation, SSA applies the concepts of Keen and Scott Morton's "Framework for Information Systems" to the analysis of the SSA Detail Model diagrams.[9] By applying this framework to the Detail Model, the structured activities that are candidates for automation can be readily identified. Moreover, the Detail Model can show where it is possible to implement Decision Support Systems. Thus, the Detail Model is a convenient medium for communicating to the user how the computerized procedures will support the business function.

Specifying the Requirements Model
Generally, the analyst proposes several alternative business solutions and documents them in separate SSA models. After the user has chosen one, the Requirements Specification Model must be prepared in detail. This model is the final product of Analysis. It represents how the business will operate when the new manual or computer procedures are implemented.

The Requirements Specification Model includes organizational requirements that are depicted in the Global Model and Function Matrix; information requirements that are described in the Flow Diagrams and Data Structure Diagrams; and procedural requirements that are detailed in the Flow Diagrams and Detail Models. Thus, it provides a comprehensive set of specifications for use during Design.

A Review of SSA Usage

SSA was originally developed as a formal technique for analysts to model both the business environment and the requirements of planned computer systems. However, as users gained experience with SSA, its potential applications increased. It has been prov-

an effective, for instance, in business improvement projects that do not necessarily involve computerized solutions. Since its introduction two years ago, SSA has gained rapid acceptance by users and analysts throughout the worldwide Exxon organization. Its acceptance is attributable to three main factors: business and systems analysts have found its notation and process usable; it can be applied to diverse business environments; and the technique can be adapted to projects of varying sizes and levels of complexity. The following studies illustrate the strengths and versatility of SSA as both a business and a systems analysis technique.

Central Purchasing System. In this application, SSA was used to specify the requirements for a large-scale purchasing system of an internationally based Exxon affiliate. Analysts used the SSA technique to interview approximately 120 users from diverse business functions. The technique helped to structure the process of gathering information and to improve communications between analysts and users. Its value was evident in the high quality of the requirements specification produced.

Refinery Blending System. In this study, SSA was applied in a technical computing environment. It was used to describe a process control system which monitored a refinery's blending operation. This case illustrates the power and adaptability of the SSA modeling notation to scientific and engineering applications.

Planning Process Study. In this project, SSA was used as a business analysis technique. It served as a documentation tool to describe the planning operation of the enterprise. Its analytical capabilities enabled it both to help identify ways of improving the planning process and to determine if the current operations could be extended to support strategic planning.

Financial Modeling System. In this case, a multipurpose financial modeling system that served many different business functions was evaluated. SSA was used to describe the business environment supported by the system and to foster communication among users. It resulted in greater understanding by each user of other users' needs, and in agreements on definitions and calculations to be used in the system.

Conclusion

SSA has been used widely in diverse business environments. Based on this experience, it is evident that the technique is valuable in several respects.

— It improves the quality of analysis and of requirements definition. SSA's systematic approach to analysis formalizes what was a very unstructured activity. It enhances the skills of experienced analysts and fosters the involvement of even the most reluctant business users during requirements analysis. Moreover, it enables analysts to prepare Requirements Specification Models that are superior to traditional narrative specifications. The resulting systems are of higher quality than before; they are more capable of representing and adapting to changing business needs.

— SSA facilitates communication between analysts and users. Their interviews are a cooperative effort to construct an SSA model of the business. The analyst does not play the role of interrogator in these interviews; rather, the participants play equal roles. In addition to improving communication during the information-gathering stage, SSA helps to describe to the users the proposed system's effect on the organization.

— SSA increases the productivity of analysts and users, enabling them to use their time more effectively and efficiently. Analysts can learn more about the business in less time. Also, requirements can be documented concurrently with data collection and analysis; SSA does not treat this task as a separate activity. Finally, there is less need to gather additional information as the project pro-

Special recognition must be given to C. M. B. Anderson (Imperial Oil, Toronto, Canada), one of the primary developers of SSA. His enthusiasm, technical expertise, and consulting expertise were critical to the successful dissemination of SSA within Exxon. I would also like to acknowledge the contributions of C. Galpin (Exxon Corp., Florham Park, New Jersey). Through his SSA teaching and consulting experience, he made significant refinements and constructive extensions to the technique.

gresses, since SSA establishes a comprehensive information base during Analysis.

— SSA provides a basis for identifying opportunities for common systems. Its modeling capabilities enable it to show the potential for generic (or multiuse) systems. SSA models of several business operations can be easily compared or contrasted, and common functions can be readily identified. It is then possible to estimate the degree of tailoring needed to apply the common system to different uses. SSA is a convenient medium for gaining user commitment to the requirements of an application that may serve several user groups.

— SSA is an important tool in project management. Its systematic process provides natural checkpoints for measuring progress. Its notation offers a means of ensuring continuity of effort and consistency in communication even when there is staff turnover. Estimates of the duration, cost, and staffing needs of the project can be obtained earlier in the system building process than would be possible without SSA.

— SSA can adapt to the way in which a system is implemented, even though its specifications are independent of it. SSA is as effective in quickly establishing a set of threshold requirements for an evolutionary or prototyped system as it is in dealing with more traditional system building approaches.

The objective of Exxon's ongoing research efforts has been to develop a compatible set of formal methodologies for building systems — from Analysis to Maintenance. SSA and PST, consistent in their approach and notation, have established a foundation upon which further research will build.

References

1
See M. Jackson, *Principles of Program Design* (London: Academic Press, 1975).

2
See B. Boehm, "Quantitative Assessment," *Datamation*, May 1973, pp. 49–59.

3
M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," *International Conference on Reliable Software Proceedings*, pp. 347–357.

4
See:
D. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, January 1977;
"SADT — Structured Analysis and Design Technique Overview" (Waltham, MA: Softech Inc., Form Nos. 9569–4, 956905, 1976);
L. Constantine and E. Yourdon, Structured Design (Englewood Cliffs, NJ: Prentice-Hall, 1979);
G. Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978);
C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques* (Englewood Cliffs, NJ: Prentice-Hall, 1979);
D. Teichroew and E. Hershey III, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, January 1977;
"Problem Statement Language Reference Summary" (Ann Arbor, MI: University of Michigan, ISDOS Project Team, Reference No. 79A51–0174–4, 1979). See also additional ISDOS documentation.

5
See:
Constantine and Yourdon (1979);
Myers (1978);
"Information Systems Planning Guide" (White Plains, NY: International Business Machines Corp., 1978).

6
See T. DeMarco, *Structured Analysis and System Specification* (New York: Yourdon Inc., 1978).

7
See Jackson (1975).

8
Ibid.

9
See P. G. W. Keen and M. S. Scott Morton, *Decision Support Systems: An Organizational Perspective* (Reading, MA: Addison-Wesley, 1978), pp. 79–98.

It is suggested is a procedure of hierarchical functional design by which programming projects can be analyzed at system, program, and module levels. It is shown that process design is made more efficient by applying Hierarchy plus Input-Process-Output (HIPO) techniques at each level to form an integrated view of all levels.

HIPO and integrated program design

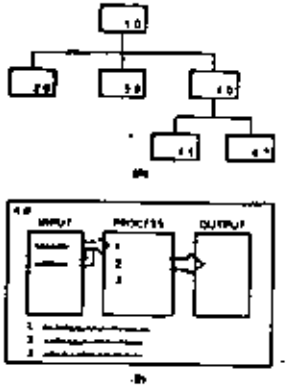
by J. F. Stay

By the mid-1970s, programming appears to be reaching a stage of refinement and cost-effectiveness such that regular business management and control methods can be applied to it. Top-down development, structured programming, chief programmer teams, structured walk throughs, Hierarchy plus Input-Process-Output (HIPO), and structured design have taken us a long way toward transforming "a private art into a public practice."[1] As a result, a body of programming knowledge and methods that are teachable and practicable has been building. This paper discusses the integration of several programming methods by means of an example.

hierarchical functional design

Most of the change in system development has been directed toward the programming effort. Although programming errors are the direct cause of many rework costs, perhaps one third of the rework ultimately can be traced back to errors in the analysis and design phases of a project.[2] Since maintenance can account for as much as seventy percent of all programming costs, more emphasis must be placed on the quality of analysis and design. Structured design and HIPO are useful techniques for organizing the application design process. This article describes how these two methods can be integrated to create a hierarchical functional design. This integrated method allows an application system to be specified from the highest functional level of a conceptual design to the lowest detailed level in a coded routine, using a single method and format. Both the concepts and the techniques of hierarchical functional design can be employed effectively throughout a development cycle in the following phases:

Figure 1 HIPO: Hierarchy plus Input-Process-Output (A) Schematic diagram of a hierarchy chart (B) Schematic diagram of an input-process-output chart



- Requirement definition.
- System analysis.
- System design.
- Program design.
- Detailed module design.
- System and program documentation.

This paper presents a basis for the thought processes involved in designing a system through the use of these techniques. Although this paper does not explain the design techniques in detail, the references provide practical help.

Two techniques for achieving functional design are the following:

- Hierarchy plus Input-Process-Output (HIPO)
- Structured design

HIPO, a technique for use in the top-down design of systems, was developed originally as a documentation tool. HIPO charts continue to serve as the final programming documentation. HIPO consists of two basic components: a hierarchy chart, which shows how each function is divided into subfunctions; and input-process-output charts, which express each function in the hierarchy in terms of its input and output. These two types of charts are illustrated in Figure 1.

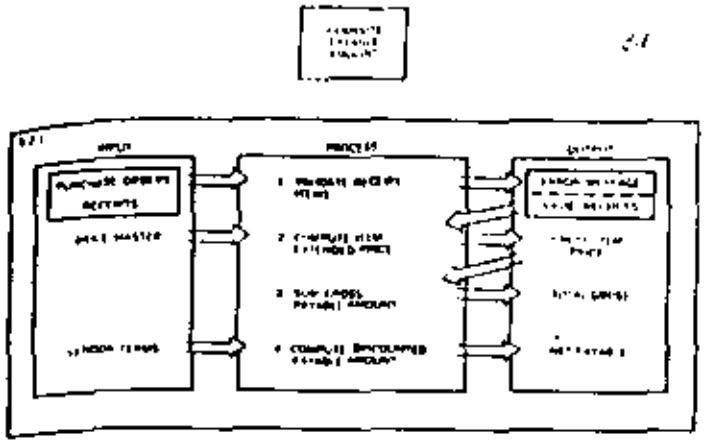
example

The HIPO design process is an iterative, top-down activity in which it is essential that the hierarchy chart and the input-process-output charts be developed concurrently, so as to create a functional breakdown. The example of COMPUTE PAYABLE AMOUNT is followed through its development process as part of an accounts payable system.

The first step is to describe a given function as a series of steps in terms of their inputs and outputs. The input-process-output chart for the example of COMPUTE PAYABLE AMOUNT is shown in Figure 2.

Having completed the input-process-chart, it is possible to move to the next level of the hierarchy. The COMPUTE PAYABLE AMOUNT hierarchy now appears as shown in Figure 3. It is now possible to develop an input-process-output chart for each of the boxes at the level shown in Figure 3. If additional definitions are required, the recommended approach is to make each line on the input-process-output chart a box on the next level of the hierarchy. This process causes the developer to focus on the level of function that is being defined.

HIPO chart for COMPUTE PAYABLE AMOUNT function



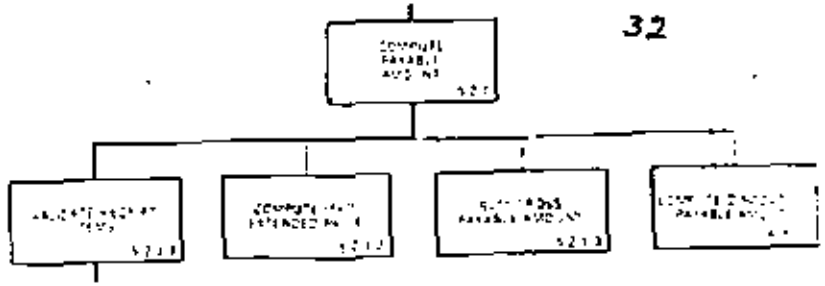
Structured design

The hierarchy plus input-process-output charting is that part of the hierarchical functional design process by which a problem description is made. The other component is structured design. Structured design[3] is a set of techniques for converting from a problem description to a functional, modular program structure. Myers[4] uses the term composite design in reference to the "structure attribute of a program, in terms of module, data, and task structure and module interfaces." This goes beyond the concept of modular design and addresses how a program module is designed, the proper scope of function of a module, and the appropriate communication between modules.

Two concepts of structured design are *module strength* (relationships within a module) and *module coupling* (relationship between modules). The way in which functions are grouped within modules determines the strength of the modules. A module may consist of a group of related functions, such as all editing functions, functions grouped according to the procedure of the problem, or all functions related to a data set.

Functional strength is the grouping of all steps to perform a single function. Although any application may contain modules that have some or all of these strengths, the objective is to produce modules that have functional strength. Functional strength does not apply only to the lowest modular level. A module may call other modules to perform subordinate functions, but if the upper-level module performs a single function and performs it completely, the module probably has functional strength. For example, the module COMPUTE PAYABLE AMOUNT might consist of the following statements (in pseudo-code):

Figure 3 COMPUTE PAYABLE AMOUNT function hierarchy



```
COMPUTE-PAY-AMT (RECEIPT), PAYABLE, RETURN-CODE):
    DO WHILE MORE-ITEMS, GET PRICE-MASTER;
        CALL VALIDATE-ITEM (RECEIPT-ITEM);
        CALL EXTEND-PRICE (RECEIPT-ITEM, PRICE);
        CALL SUM-AMOUNT (PRICE, TOTAL-PRICE);
        CALL COMPUTE-DISCOUNTED (TOTAL-PRICE,
        DISCOUNT-RATE, PAYABLE);
    END DO;
END
```

This module performs very few functions by itself. However, it transforms one input (RECEIPT) into one output (PAYABLE) completely; at the same time it does no unrelated processing. Therefore, this module is said to have functional strength.

Interactions between modules, termed *module coupling*, may be as varied as interactions within a module. The extreme of module coupling exists when one module directly modifies an instruction in another module.

The preferred relationship between modules is *data coupling*. In data coupling, each module simply passes application data, usually as parameters to the next lower-level module. Use of artificial switches and indicators is avoided. When a calling module passes switches, indicators, or other control information to another module, these items must only communicate the status of the calling program. The calling program should not assume that it knows what the called program will do, based on this control information. A serious problem in program maintenance results when a simple change to a module changes the meaning of an item of control that has an unsuspected effect on the logic flow of one or more other modules.

This paper does not treat structured design in depth but only with sufficient detail to carry the concepts of functional strength and data coupling into earlier stages of the design process. The reader may find References 3 and 4 to be of valuable assistance in module design.

Hierarchical functional design addresses essential processes that may be applied to the application analysis and design tasks. Hierarchical functional design applies the design constructs of functional strength and data coupling to the functional decomposition and graphic techniques of HIPO to provide a single methodology that allows an application design to develop in an orderly manner from a clear statement of the requirement to an intelligible, well constructed set of application functions. A system that is designed through the use of hierarchical functional design is implemented in a top-down manner. Modules should have a single entry point and a single exit point; they should be small; and they should use the SEQUENCE, IF-THEN-ELSE, and DO-WHILE concepts of structured programming. Hierarchical functional design can be used in all phases of the development cycle, and thereby provide a visable system that is suitable for a design walk through. The chief programmer team concept is also supported, since functional breakdown with clearly defined interfaces allows modules to be delegated to developers or to other teams, with the common understanding that is required for programs to integrate properly.

Hierarchical functional design employs the following three design concepts:

- A functional design in which the computer solution is structured in terms of the user's function.
- An iterative process in which each level of design is validated against the level above it.
- Conceptual levels of design, in which each level emphasizes a particular aspect of the problem solution.

A computer system can be viewed as a single function that can be divided (or decomposed) into a hierarchy of sets of successively lower-level functions until the elemental functions are described. An understanding of the meaning of the term "function" is necessary for further discussion. *Function* can be defined as an action upon an object, or, for our purposes, the transformation of some input data to some return data.[3] A statement of function describes what is done rather than how it is done. Since a function is also singular, it is defined with a simple declarative statement that consists of only one verb and one object. Both the verb and the object may be conditional.

A function should also have the characteristics that are defined for structured design. A function should be completely defined in one place, and relationships among functions should be primarily data relationships. Thus the concepts of structured design are valid for designing systems as well as modules. Functional

design, then, consists of stating what is to be done in terms of data in and out. A high-level functional statement is reduced to a set of more detailed low-level statements, in a verb-object format. The set of lower-level statements must equal the function of the higher-level statement. In the accounts payable example that is used in this paper, the high-level function is COMPUTE PAYABLE AMOUNT. This function is stated in terms of its input data (purchase receipt) and its output (net payable amount). In this case, the output may simply be passed to another function. The function COMPUTE PAYABLE AMOUNT is then reduced to the following four functional statements:

VALIDATE RECEIPT ITEMS
COMPUTE ITEM EXTENDED PRICE
SUM GROSS PAYABLE AMOUNT
COMPUTE DISCOUNTED PAYABLE AMOUNT

Each of these statements can then be expressed in terms of its input and output data. This set is an explicit statement of the steps required to perform the function COMPUTE PAYABLE AMOUNT.

This is only one example of the way in which a function may be subdivided. The structuring of subfunctions requires analytical skill and imagination, and each analyst may define the subcomponents of a function slightly differently. It is important, however, that the definition of a function determine the functions that are subordinate to it. The function COMPUTE PAYABLE AMOUNT could not legitimately have a subordinate function that, for example, updates the inventory balance.

functional design — iterative process

The design of an application should be an orderly growth process from inception to implementation. During development, frequent reviews should be conducted so that a given design always meets its objective. Design has usually been done at least twice before a system is complete and running. First, a functional design has been made to provide an understanding between the user and the programming department. Then a logic design has been made, from which programming could proceed. With hierarchical functional design, the function is the logic, and redundant effort may thus be avoided.

Hierarchical functional design is an evolving, top-down process. The first step is a translation of a statement of need into a functional statement of system objectives. As information about a required system is gathered, that information is organized according to the functional structure. The first statement should contain display screen formats, report layouts, perhaps a two-level hierarchy chart, and a single level of HIPO charts. The analyst should walk through these charts with the customer to ver-

fy that this level of design conforms with the requirement. As the process of design moves to areas such as file access methods, record layout, and message traffic, definitions in successively lower levels the hierarchy may cause upper levels to change. This is typical of the program development process and is the reason for continuing discussion with the customer. This is the iterative process – the refining of the higher levels of design as the more detailed levels are developed. As the iteration of detail progresses downward, the impact on the top-level design should become minimal. Because of the successive iterations of assessing the upward impact of design decisions, the result is a stable, intelligible, and maintainable design.

Levels of the design hierarchy

As an application is divided into functions and each function in turn is subdivided, the hierarchy proceeds toward greater detail. All levels of the system are described as functions, and can be grouped into three categories, proceeding from the broadest to the finest level of detail, as follows:

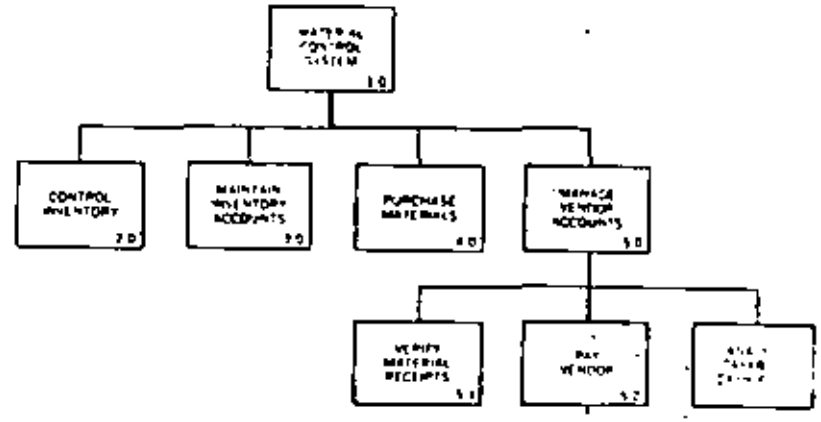
- System
- Program
- Module

These three levels are conceptual, and are not a physical part of the design. Each conceptual level may represent multiple levels on the hierarchy chart, and any given box on a chart may be both the bottom of one conceptual level and the top of the next level.

system level

The system level of the hierarchy contains the major component parts of the application, and is the view that a department manager might have of the application. A system might contain multiple system levels: Accounts payable is a component of a material control system, and in turn, the subsystems of accounts payable itself would be contained within the system level. This level of hierarchy is started by structuring the original statement of requirement for an application. The analysis phase of a project may result in a system-level hierarchy such as that in the example in Figure 4. Each box in Figure 4 can be stated in functional terms, and can be represented by a HIPO chart. Although the terms of the user may differ somewhat from those on the chart, the functional statement should be used because it is more explicit than the common term. For example, the term ACCOUNTS PAYABLE may be simply a subset of the general ledger, or it may be a complete system for managing the payment of vendor accounts. The functional term MANAGE VENDOR ACCOUNTS makes the objective of this application more clear.

Figure 4 System level of the accounts payable application



The system level normally does not include the representation of any executable computer instructions, but rather it provides a conceptual view of the application. Input and output are defined in terms of forms, files, and reports, which are a user's view of the data. If it appears that the user manager's view of the structure of an application does not provide the basis for program design, it should be mentioned that one of the primary objectives of functional design is to have a single view of the application that represents both the user's requirement and the program design. Although functions defined at the top level may be grouped differently for program design, they should all still exist with the same basic relationships in the computer implementation. This is the key to building systems that can be readily maintained and enhanced.

program level

The program level of the hierarchy shows the highest level of segmenting of the computer system. The program level may also be characterized as the end-user level, and represents the level of tasks initiated by a terminal operator in an interactive application, or batch programs in a batch application.

In the accounts payable application, consider the boxes below MANAGE VENDOR ACCOUNTS as tasks or programs. An example program level (one of the boxes in the accounts payable application) is shown in Figure 5. The program or task level is a result of the general design phase of a development project. Input and output for this level are usually defined as records or groups of records, messages, and report lines. Since this level normally represents executable computer instructions, it is recommended that program and module names be assigned to the boxes.

module level

Detailed program design occurs at the module level. Since design is an iterative process, detailed module design may expand

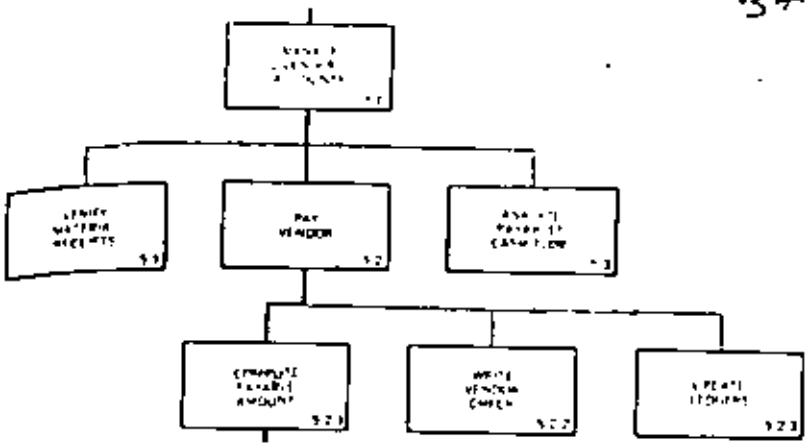
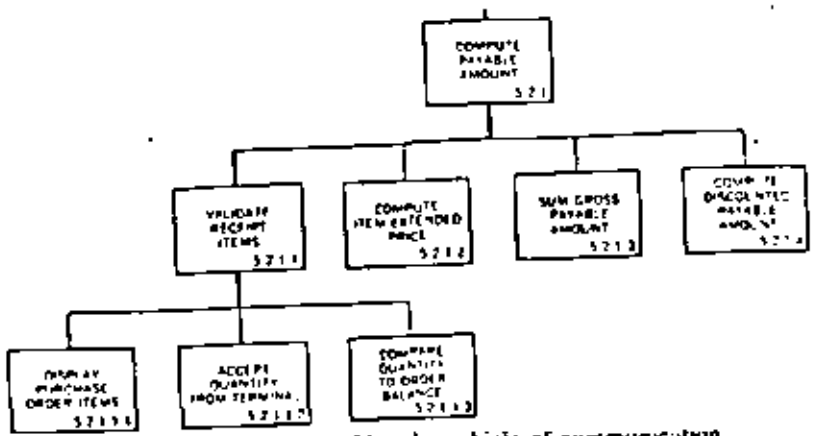


Figure 6 Module level of the accounts payable application



flaws or required restructuring of the more general design. It is essential that the upper-level HIPO charts be revised and revalidated before continuing with the design process. Design modification may be required when a low-level change causes the higher-level design to do something different from that which the user requires.

The module level represents an executable segment of program code that is usually compiled as a unit. This unit of code is typically called an "object module" to distinguish it from a "load module," which may be created by linking several functional modules together. At the module level of the hierarchy, the design is sufficiently detailed that program code can be written directly from the design. In the accounts payable application, two levels of modules are shown in Figure 6 below COMPUTE PAYABLE AMOUNT, which can be related to the program level in Figure 5.

The module level is the product of the specification phase of development. Input and output at this level are fields of data or parameters from or to other modules. The module level should represent a functional statement that can be completely grasped within a normal attention span. When translated into executable code, a module should usually contain fewer than fifty lines of structured high-level language statements. The HIPO chart at the module level may contain structured English (pseudo-code) statements to explain complex logic. In addition, where necessary, the extended description section of the HIPO chart may provide implementation notes as discussed in Reference 5.

The purpose of these conceptual levels is to reflect the objectives of users of the documents. The system level must be stated in terms that are relevant to user management. The module level is organized in such a way as to allow the programmer to write code. The program level provides the vehicle of communication between the system level and the module level by giving detail to the user and a higher-level view to the programmer.

Hierarchical functional design in a virtual system

The discussion thus far has considered the structure of an application as an aid in the design of an intelligible, maintainable system. A current major concern in system design is that of providing efficient performance of interactive applications in a virtual system environment. Performance tuning in a virtual system is a complex science that involves relationships among hardware, system software, and application design.[6,7] Optimizing the performance of the application programs alone does not result in an efficient system. A conscious effort of tuning all the components that affect performance is required because programmers, for example, often attempt to write efficient—sometimes complex—code without regard for the way in which the modules may ultimately affect performance.

When viewing the structured design of an application, one can see readily that functional decomposition does not reflect the performance requirements of a transaction driven application. Hunter,[3] for example, makes clear the issue of the compounded effect of excessive paging. He defines the working set as the twenty percent of the code that does eighty percent of the work, and advises that the "working set should become the focus of application tuning." The design process, if correctly executed, can produce modules each of which requires less than a single 4K byte page of storage. On that basis, the task of application tuning becomes an effort of identifying the most active modules, rewriting those modules for efficiency (if necessary), combining modules into logically related pages, and fixing active pages in main storage.

performance tuning may, in extreme circumstances, require the surgical modification of a few critical modules. Such modifications may include changing a CALL (transfer of control to a routine) to a simple inclusion or even the integration and restructuring of modules. It must be clearly understood that only a very few modules ever seriously affect the performance of most systems. Thus, if an application is structured, the necessary tuning can be done consciously with proper control. With this perspective, even extreme coding techniques may be justified for those few modules that must be efficient to avoid performance degradation.

Concluding remarks

The improvement of the system development process requires innovation in two areas: the development of a discipline that involves a set of structured techniques, and an understanding of the theories on which that discipline is based.

Significant progress has been made in developing a discipline that, in time, should make program design and implementation an engineering skill. Structured programming provides basic building blocks for code development, much as electronic circuit development can be based on a set of predesigned, basic electronic components. HIPO and structured design are first steps in bringing that type of discipline to the design stage of application development.

It has been the intention of this article to address the following mental processes:

- Identification of function.
- Functional decomposition.
- Iterative design.
- Module relationship.
- Delayed performance optimization.

By applying the concepts of hierarchical functional design to the disciplines of HIPO and structured design throughout the analysis and design process, several of the following possible benefits are typically realized:

- User understanding and agreement on functional content are made easier.
- Missing or inconsistent information is identified early.
- Functions are discrete and are therefore more easily documented and, if necessary, modified.
- Documentation is accomplished with a single effort rather than multiple efforts at different stages of development.

- Module interfaces are simple and therefore reduce the probability of logic errors.
- The resultant design supports structured, top-down coding.
- Maintenance and enhancement are more transferable because the system can be easily understood at all levels.

Since these processes are ways of thinking about the design activity, it is often difficult to measure objectively the effect of using this knowledge. By applying these principles to a development project, however, one becomes aware of their value.

CITED REFERENCES

1. H. D. Mills and F. T. Baker, "Chief programmer teams," *Datamation* 19, No. 12 (December 1973).
2. P. Moody and R. Perry, "Application development cycle problems," *Proceedings of Guide 38*, Miami Beach, May 18–23 (1974).
3. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal* 13, No. 2, 115–139 (1974).
4. G. J. Myers, *Reliable Software Through Composite Design*, Mason/Charter Publishers, New York, New York (1975).
5. John J. Hunter, "Rethinking application programs, key to VS success," *Computerworld*, March 25, 1975.
6. J. G. Rogers, "Structured programming for virtual storage systems," *IBM Systems Journal* 14, No. 4, 385–406 (1975).
7. H. A. Anderson, Jr., M. Reiser, and G. L. Galati, "Tuning a virtual storage system," *IBM Systems Journal* 14, No. 3, 246–263 (1975).

GENERAL REFERENCES

1. *HIPO—A Design Aid and Documentation Technique*, Order No. GC20-1851, IBM Corporation, Data Processing Division, White Plains, New York 10604.
2. *Structured Programming Independent Study Program*, Order No. SR20-7139, IBM Corporation, Data Processing Division, White Plains, New York 10604.
3. J. D. Aron, *The Program Development Process, Part I, The Individual Programmer*, Addison-Wesley Publishing Company, Reading, Massachusetts, 31–36 (1974).

Reprint Order No. G321-5031.

The *Second* Structured Revolution

Edward Yourdon
YOURDON inc.
1133 Avenue of the Americas
New York, New York 10036

March 1979

1. INTRODUCTION

After nearly ten years of discussions at computer conferences and in trade journals, nearly everyone in the data processing field has at least *heard* of such technologies as structured programming, structured design, structured analysis, top-down implementation and structured walkthroughs. Indeed, many programmers and analysts will tell you confidently that they *understand* the structured techniques, and that they are faithfully *using* the techniques.

Alas, it just isn't so. The structured techniques are *not* as widely understood or used as one might imagine from reading the popular literature. A recent survey in *Datamation*, for example, indicated that only a small fraction of the major EDP organizations in southern California were making consistent use of structured programming and design.

Even worse, many large organizations are finding that the large sum of money they spent attempting to convert their staffs to a "structured" approach has been wasted. After all the fanfare of the "structured revolution" — and after spending a considerable amount of time, energy and money on training, textbooks and guest lectures by the prophets of structured programming — many organizations are finding that their people are still developing the same expensive, bug-ridden, slipshod, unmaintainable software that they were developing before structured programming was introduced.

Why? Is it because our programmers and analysts are too stupid to learn new techniques? Is it because our programming languages and operating systems don't support the new techniques? Or is it just that the Forces of Evil are determined to prevent us from ever being able to develop quality software?

No! The real problem is *management*. It's my opinion that management is to blame for the failure of the structured revolution. And only management can ensure that the *second* revolution — a revolution that is now beginning in several EDP organizations — will be successful.

The purpose of this paper is to explore the *failure* of the first revolution in more detail — so that organizations that are just beginning to think about structured programming can avoid those failures. Then we will present a battle plan for the second revolution, a plan for implementing structured techniques economically and successfully.

2. AN OVERVIEW OF THE STRUCTURED TECHNIQUES

Let's begin by summarizing the structured techniques themselves. Obviously, a one-paragraph summary won't explain all of the technical details of each technique; you may find it helpful to refer to some of the standard reference texts on the subject for more details.

Structured analysis is a collection of graphical tools that help the systems analyst document the *functional specifications* of a system. Instead of the classical *narrative* specification — which, like a Victorian novel, tends to be monolithic, verbose, redundant, ambiguous and boring — structured analysis allows the analyst to portray the user's system as an abstract, partitioned, top-down "model" of the system-to-be. The primary tools of structured analysis are data flow diagrams, data dictionaries, data structure diagrams and structured English.

Structured design is a collection of guidelines and strategies that help the designer select the models and the module interfaces that will most economically implement a software system that has already been well-specified. Structured design consists of documentation techniques such as *HIPO* and *structure charts*. It also consists of "evaluation criteria," notably *coupling* and *cohesion*, which help the designer distinguish between good designs and bad designs. And structured design also includes "cookbook strategies" (e.g., transform analysis and data-structure analysis) that help the designer systematically generate good designs for common types of problems.

Structured programming is a coding discipline based on the concept (first published in the mid-1960's) that all program logic can be built from combinations of:

- "sequential" instructions — e.g., MOVE, ADD, SHIFT, etc.
- IF-THEN-ELSE
- DO-WHILE

Often referred to as GOTOless programming, structured programming probably the oldest of the structured techniques and is now taught in many universities.

Top-down implementation is a strategy for building software systems that have been specified and designed. In sharp contrast to the classical "bottom-up" approach (in which modules are tested first, followed by program testing, subsystem testing and system testing), the top-down approach calls for implementing high-level "executive" modules *first*, with the lower-level modules implemented as "stubs" or "dummy modules" (e.g., modules which return constant outputs or which exit without doing any processing). The top-down approach has the advantage of allowing the EDP personnel to demonstrate a "skeleton" version of the system to the users at an early stage, so that the users can see if they are getting the system they really want.

Walkthroughs are generally thought of as a peer-group process for reviewing the products of structured analysis, structured design and structured programming. Managers, "big bosses" and other outsiders are generally not invited to walkthroughs since their presence often results in a review of the producer rather than the product. The primary purpose of the walkthrough is to ensure the quality and correctness of the product (whether it be a functional specification, a design or a page of COBOL coding); secondarily, the walkthrough approach serves as an excellent training device, and it makes the project less vulnerable if a key technician leaves in the middle of the project.

3. WHY DID THE FIRST REVOLUTION FAIL?

Before we can explore successful strategies for introducing structured software development techniques, we need to explore in more detail the reasons why they weren't introduced successfully the first time around. The following reasons seem to be most important.

Inadequate selling of the techniques

In many companies, most people in the EDP organization remain blissfully unaware of all new technological developments: they're quite happy to keep developing systems the way they have been for the past umpteen years. So when the resident technical hot-shot comes back from an ACM conference with some new ideas about structured programming, he's likely to be booted down by his colleagues. When an aggressive young project manager returns from a GUIDE conference with some interesting case studies about successful uses of structured analysis, his colleagues will mumble something about, "Well, that's fine for XYZ company, but it won't work here!"

Indeed, it's even worse in some organizations, where the practice of "sending out the scouts" is used to examine new technologies. The "scout" often turns out to be the worst of all salesmen in terms of convincing his organization to begin using something that he's learned about in a conference or a symposium — indeed, he's often so brash, so intellectually superior and so downright arrogant that the only way his organization can stand him is to send him away regularly to such meetings!

Even if the scout is reasonably diplomatic and articulate, he may have great trouble selling the structured techniques to his colleagues simply because, by virtue of working within the organization, his colleagues view him as not expert enough to make them change their way of doing things.

Inadequate training

In the organizations where "structure," as it's often known, does gain some degree of acceptance, massive training programs are often begun — but as my distinguished colleague Gerald Weinberg points out, it was often done with a "sheep dip" approach: large numbers of programmers and analysts are herded into an auditorium, where they are, in a sense, dipped in a bath of structured snake-oil. That is, an outside "expert" (who may be little more than a salesman for his firm's products or services) is asked to give a short half-day presentation on "Everything You're Ever Likely to Need to Know About All That Structured Stuff" to the assembled group — whereupon they are herded back to their desks and told that their development schedules have just been cut in half because the expert has said that "structure" doubles programming productivity!

Even when the EDP organization provides thorough, comprehensive training, there's no guarantee that the students have learned what they were taught. And, more important, there is no guarantee that they will *use* what they learned. As a teacher, I have had far too many situations where a student came to me at the end of a course, shook my hand and said, "Well, that was sure a nice course — now I can get back to drawing flowcharts in my maintenance shop again!"

Inadequate management support and follow-through

Ultimately, the problems mentioned above have to be described as "management" problems: *management* has got to be involved in the "selling" of structured systems development techniques, and *management* has got to be aware of the need for proper training.

But even more important than the training and the consulting (both of which can be accomplished with a great deal of fanfare and a great expenditure of money) is the *follow-through*. There are a number of large EDP organizations that did a reasonably good selling job when the structured techniques first became popular in the mid-1970's, and that also provided a substantial amount of high-quality training. But when it came time to put the techniques to work on *real* projects, things began to fizzle. It's not hard to imagine the kinds of situations that develop:

- A project manager learns about the structured techniques *after* his project team has finished its analysis and half of its design — so he decides that he might as well ignore *all* of the structured techniques, even though top-down implementation and walkthroughs would be eminently practical.

- A project manager decides that the project he is about to undertake is far too sensitive and far too critical to risk using "new-fangled" techniques.
- Conversely, a project manager decides that his project is too small and too simple to warrant using such "high-powered" development techniques.
- The project manager begins using the new techniques but then runs into "mid-project panic": because of the learning curve associated with the introduction of *any* new set of ideas, the project team thrashes around and spends a considerable amount of its time arguing about esoteric theory — so the project manager, in a moment of panic, decides to abandon the structured techniques and return to the more familiar methods of developing software.

4. HOW TO DO IT RIGHT

If these are the problems, then how can an EDP organization introduce the structured techniques — or, for that matter, any other new technology — successfully?

To a large extent, the answer is: *approach it the same way you would approach a "real" EDP project.* Just as a "normal" project requires distinct activities of analysis, design, implementation and testing, so the introduction of structured development techniques requires the same activities.

An analogy might help illustrate the point. A competent EDP professional would never dream of walking into a user organization and thoroughly disrupting its way of doing business with the introduction of a new computer system — not without a great deal of planning, analysis and design. And since the EDP organization is (when viewed from the outside) a *business*, it should be approached with the same caution that one would use when approaching a user's business.

What's needed, then, is the following:

- A dignified "selling" effort to convince the EDP staff that the structured techniques are, if nothing else, at least worth investigating.
- A formal analysis activity.
- A formal design activity.
- A formal implementation activity.
- A formal testing activity.

Each of these activities is discussed in more detail below.

5. THE SELLING OF STRUCTURED X

Don't be misled into thinking that "selling" is unnecessary or unimportant. The larger the organization, the more likely it is that there will be pockets of resistance (or downright reactionary ignorance!). If your local hardware vendor has already done a selling job for you, that's fine — but if not, make sure that the key people in your organization are aware of the virtues of the structured techniques, and what their benefits are likely to be.

In most organizations, you'll find that you have to approach several distinct *levels* of people as you do your selling job. The boundaries between these levels are somewhat fuzzy, but basically they are as follows:

- Top-level EDP management and, on occasion, managers above the EDP organization. This might include people like the VP of Finance, or the Director of MIS or the Manager of Computers & System Development, etc.
- Middle-level managers — e.g., first-level team leaders, second-level project leaders, etc.
- Technicians — i.e., the programmers and analysts who will be most directly involved in the day-to-day *use* of the structured techniques.

Top management, in my experience, is usually not involved in the day-to-day crises of software development. If a project is behind schedule, *they* won't be spending their nights and weekends in the computer room with the harried crew of programmers. So they may not feel the same sense of urgency about the introduction of new development techniques that the first-level managers do.

Not only that, they're probably totally uninterested in — and unaware of — the technical details of structured analysis, structured design and structured programming. Indeed, one of the most frustrating experiences I've ever had is trying desperately to explain such technical issues in language simple enough and "jargonless" enough that an insurance vice president could understand it, only to have him say, "Gee, that sounds awfully simple — in fact, it sounds like plain old common sense. Are you sure our people haven't been doing this structured stuff all along?"

So skip the technical details. Concentrate instead on arguments of *economics* — i.e., how much money will structured programming save the organization? How will maintenance costs be reduced? How many fewer of those long-haired, unwashed programmers will we have to hire this year if the new techniques are introduced? After all, if it's a high-level manager that you're talking to, the economic — that is to say, the *business* — ramifications of the structured techniques are what concern him the most anyway.

Middle management is where you'll have the most trouble selling the techniques, *particularly* if you try to sell the techniques in a negative way! It's very easy for a veteran project manager to get the impression that the introduction of structured analysis or design or programming is an implied criticism of the way he's done his job for the past fifteen years. And if he gets that impression, he'll be able to come up with a hundred different reasons for why "It'll never work here!"

When I use the term "middle-level manager" I have the image of someone who has come up through the ranks — that is, someone who was once a programmer or systems analyst. That can be both good and bad. Such a manager ought to be able to understand the technical aspects of structured development techniques, *but* there's always the danger that he will try to think about the structured techniques in the context of RPG and the IBM 1401 — and that's when you'll start getting complaints like, "That stuff sounds too inefficient — why I remember that a subroutine call used to take 6.573 microseconds . . . "

There's one other thing about the middle-level manager: for the most part, he *is* involved in day-to-day crises of software development. More to the point, he's plagued by pressures of deadlines. He may agree with you that the new techniques can reduce maintenance

costs — but right now he's faced with a deadline that he can just barely meet if he applies all of his conventional techniques. And he's not allowed to add any people to the project; and management will have a coronary if he suggests slipping the deadline; and besides, all the money in the training budget has been allocated to learning the mysteries of relational data base systems.

What can be done in a situation like this? Realistically, the answer is — not much. In the long run, the support and the budget and the extra people and the extended deadlines have got to come from higher levels of management. It's possible, of course, for the middle-level manager to gamble: he can try to pick a medium-sized project that doesn't have all the cards stacked against it — i.e., one that can always be rescued by what my colleague Tom DeMarco calls "the time-honored tradition of unpaid overtime." If you're a middle-level manager, though, and you decide to do something like this, you should be very aware that it's a gamble. For the first several weeks (if even months), when everyone expects your people to be busily coding and they're doing nothing but arguing about data flow diagrams and functional cohesion, about the only thing you'll be able to say is, "Trust me . . . it will all work out. Trust me."

Technicians also have to be sold — though my experience is that a great number of them are already convinced. Many of them have complained for years that "management" (that great nemesis, second only to users as a source of frustration) never has time to do the job right the first time around, but always has time to do it twice.

Of course, there will be senior technicians — the grizzled veterans who have been programming since 1952 — who will have many of the same reactions as the middle-level project managers: if it didn't work on the IBM 650, it certainly won't work now! And there will be moderately experienced technicians — with 2 to 5 years of experience — who will accept the structured techniques intellectually, but whose brains have been so badly warped by years of terrible programming languages (e.g., COBOL, FORTRAN, PL/I, RPG . . .) that they'll never *really* specify, design or code a structured program.

Indeed, it may well be the trainees — those with virgin minds — who are our only salvation. Obviously, trainees are (or should be) humble enough to do what they're told — and they can be trained to do it right from the very beginning. *If* — and this is a very large if!! — they are trained properly. It's surprising to see how many organizations are still teaching "basic" courses in programming, design and analysis that are riddled with ideas and techniques that are grossly obsolete — only to follow this up with "advanced" courses that preach all of the modern structured techniques. The student spends the first half of such advanced courses being asked to *unlearn* a substantial amount of material that he learned in the basic course — something that's obviously unpleasant and difficult to do.

One last suggestion about the selling of structured X: *do it top-down!* If top management doesn't support the new development techniques, there's not much point training the technicians. Except in places like Iran, grass roots revolutions don't seem to be in vogue these days.

6. THE ANALYSIS ACTIVITY

Perhaps the most important aspect of introducing structured development techniques is a formal *analysis* of the way the EDP organization develops software. In other words, the analysis activity should be attempting to answer the following questions:

- Do we have a problem in our EDP organization today?
- If so, what is the nature of the problem? Is it political? Is it hardware-oriented? Is it associated with the maintenance of systems developed 15 years ago?
- How serious is the problem? Or, to put it in a more positive light, how much money could be saved by introducing the structured techniques?
- How long would it take to convert the organization to the structured techniques? What is the "return on investment"?
- What are the risks? Is it possible that the structured techniques could destroy the organization? Will all of the newly trained programmers immediately quit and get better-paying jobs elsewhere?

In effect, this amounts to a "feasibility study" and a detailed examination of the way the EDP organization is presently doing business. Obviously, if the organization has no problems (either real or perceived), there's not much incentive for introducing the new techniques. If there is a problem, and if it's perceived as a serious problem, and if it appears to be related to the methods for developing software — *then* there may be a good argument for introducing the new techniques. But the argument should be made with the same careful analysis that one would use to convince a user organization to install a new-fangled computer system.

7. THE DESIGN ACTIVITY

After the analysis has taken place, the next step is *design*. Basically, this involves developing a *plan* for implementing the techniques — as opposed to a "shotgun" approach of training everyone in sight, issuing "structured edicts" that will probably be ignored anyway, etc. The design activity should address the following kinds of questions:

- Which of the structured techniques should be implemented first?
- Which parts of the organization should be exposed first?
- What kind of training is appropriate? For whom?
- What kind of pilot projects should be used to experiment with the new techniques?
- How can we manage and control the process of converting the organization to the new techniques?
- How can we measure the impact that the structured techniques will have on the productivity of the EDP organization?
- How can we strike a compromise between the deadline pressures of today and the long-range benefits of the structured techniques?

Obviously, these are not simple questions — and as you can imagine, there are no "pat" answers for any of them. However, my own experience with a number of organizations leads me to offer the following suggestions:

1. *Informal walkthroughs are a good way to begin.* In addition to the "obvious" benefit of reducing errors and increasing the quality of specifications, designs and code, the introduction of walkthroughs can help ensure some *consistency* in the use of structured techniques by your staff. This is particularly important when a group of programmers and analysts all begin using structured programming, design or analysis at the same time — there is a great danger that each technician will interpret differently the techniques that he has learned from a textbook, a video training course or a classroom training experience.

2. *Don't begin with structured coding — start with analysis or design.* There's an obvious reason for this suggestion: with good analysis and design, you can tolerate mediocre code; on the other hand, brilliant code can't save a bad design or a fuzzy specification. There's a more important reason for making this suggestion, though: assuming that it takes one to two years (or more) to introduce the new techniques, there's some danger that the EDP organization may "run out of steam" after six months and stop introducing new techniques. If such is the case (and it's happened in a number of organizations, either because of economic considerations, a change in management or other subtle political reasons), then it would be sad to find that the only thing that had been accomplished was an improvement in coding techniques.

3. *Don't introduce too many new techniques at once.* Attempting to get a large group of people to adopt a dozen new techniques at once is a sure-fire way of guaranteeing that none of the techniques will be introduced successfully. Begin with just one or two new techniques — e.g., structured analysis and walkthroughs. Then move on to structured design and top-down implementation; then perhaps structured programming . . .

4. *Choose one or more medium-sized "safe" pilot projects.* The notion of a pilot project is familiar to most EDP organizations — that is, a formal experiment to see if a new technology really works. In our experience, the best pilot projects have been at least six person-months long; they have been projects that are "real" and visible to the organization. But at the same time, they have not been projects whose failure would bankrupt the organization! Indeed, the very best kind of pilot project, in many cases, is a "conversion" — i.e., rewriting an old system that was about to collapse anyway. Among other things, such a pilot project offers a basis for comparison (admittedly a somewhat biased comparison, but better than no comparison at all) between the old way of doing things and the new way of doing things.

5. *Groom some internal "structured gurus" for ongoing consulting assistance.* Another advantage of the pilot projects mentioned above is that it accomplishes *this* suggestion. It provides the organization with a group of people who have really *done* structured analysis, design and/or programming, as opposed to people who have done nothing more than just read about it in a book. This, in my opinion, is essential to the success of installing the new techniques in a large organization.

6. *Establish a group with responsibility for coordinating the introduction of the new techniques.* This can be the training department (though they often have zero credibility in the EDP organization), or the standards department (who, unfortunately, often have *negative* credibility in the organization) or a special task force reporting to an appropriately high level in the EDP hierarchy. But the main point is obvious: The techniques won't be introduced by themselves — and, left to their own devices, each programmer or analyst will be-

gin practicing his own interpretation of that subset of the structured techniques that he wants to introduce. And nothing more!

8. IMPLEMENTATION

By implementation, I mean the process of actually training the staff, doing the "grunt work" of rewriting appropriate standards and doing the messy managerial/political work of enforcing the new methods of analysis, design and coding. Most of the actual work will be obvious, once the analysis and design activities have been completed (note that the same thing is true in a "real" EDP project).

One word of caution is in order though: the implementation process may go on forever. This is partly because of the turnover and staff growth that the EDP organization generally faces. It may also happen for another reason: if most of the programmers and analysts are engaged in pure maintenance work, there may be little opportunity (or motivation) to train them in structured analysis or structured design. Thus, if the new techniques are introduced solely to technicians working on *new* development projects, it may be five to ten years before it has filtered through the entire organization.

There is one other point to consider, particularly if the introduction of the new techniques is going to stretch over a period of five years, and that is: *the techniques themselves will continue to expand and change over the next several years.* We can certainly expect that structured analysis and structured design will be refined, enlarged and improved over the next several years -- so the kind of training done in 1984 will most likely be different from the kind of training done in 1979 or 1980.

9. THE TESTING ACTIVITY

The activity most often forgotten or ignored by EDP organizations installing the structured techniques is *testing*. The basic objective of the testing activity is to continually monitor the organization to find out:

- whether the technicians have actually learned the techniques,
- whether they are using the techniques in a uniform fashion,
- how much improvement there has been.

This implies, then, that there needs to be a "quality assurance" group or an auditing group continually inspecting the methods used by the organization.

10. SUMMARY

Despite some of the fanfare and publicity in the EDP journals, the structured techniques are not "magic." Indeed, one could argue that they are nothing more than "common sense" — but the crucially important point is that the structured techniques represent a *standardized* common sense. And besides, as Will Rogers once said, "Common sense isn't common."

Whether they represent common sense or black magic, there is increasing evidence to confirm that structured analysis, design and programming can *substantially* increase productivity, maintainability and reliability of EDP systems. Certainly, any EDP organization whose existence depends heavily on the quality of its software is going to find its existence seriously threatened in the next decade if it does not adopt modern development techniques.

But the question, as we have seen in this paper, is *how* to introduce the new techniques. A small organization of geniuses can — perhaps! — implement the structured techniques on an *ad hoc* basis. But a large organization, with hundreds of programmers, can literally put itself out of business if it doesn't have a *plan* for implementing the techniques.

A summary of progress toward proving program correctness

by T. A. LINDEN

National Security Agency
Ft. George G. Meade, Maryland

INTRODUCTION

Interest in proving the correctness of programs has grown explosively within the last two or three years. There are now over a hundred people pursuing research on this general topic; most of them are relative newcomers to the field. At least three reasons can be cited for this rapid growth:

(1) The inability to design and implement software systems which can be guaranteed correct is severely restricting computer applications in many important areas.
(2) Debugging and maintaining large computer programs is now well recognized as one of the most serious and costly problems facing the computer industry.
(3) A large number of mathematicians, especially logicians, are interested in applications where their talents can be used.

This paper summarizes recent progress in developing rigorous techniques for proving that programs satisfy formally defined specifications. Until recently proofs of correctness were limited to toy programs. They are still limited to small programs, but it is now conceivable to attempt to prove the correctness of small critical modules of a large program. This paper is designed to give a sufficient introduction to current research so that a software engineer can evaluate whether a proof of correctness might be applicable to some of his problems sometime in the future.

THE NATURE OF CORRECTNESS PROOFS

Given formal specifications for a program and given the text of a program in some formally defined language, it is then a well-defined mathematical question to ask whether the program text is correct with respect to those specifications. The mathematics necessary for this was originally worked out primarily by Floyd[1] and Manna.[2]

It must be made clear that a proof of correctness is radically different from the usual process of testing a program. Testing can and often does prove a program is incorrect, but no reasonable amount of testing can ever prove that a nontrivial program will be correct over all allowable inputs.

Example

The approach to proving programs correct which was developed and popularized by Floyd is still the basis for most current proofs of correctness. It is generally known as the method of inductive assertions. Let us begin with a simple example of the basic idea. Consider the flowchart in Figure 1 for exponentiation to a positive integral power by repeated multiplication. For simplicity, assume all values are integers. I have put assertions or specifications for correctness on the input and output of the program. We want to prove that if X and Y are inputs with $Y > 0$, then the output Z will satisfy $Z = X^Y$. This assertion at the output is the specification for correctness of the program. The assertion at the input defines the input conditions (if any) for which the program is to produce output satisfying the output assertion. Note that the proof will use symbolic techniques to establish that the program is correct for all allowable inputs.

The proof technique works as follows: Somewhere within each loop we must add an assertion that adequately characterizes an invariant property of the loop. This has been done for the single loop flowchart of Figure 1. It is now possible to break this flowchart into tree-like sections such that each section begins and ends with assertions and no section contains a loop. This is

...wn in Figure 2 if one disregards the dashed-line ...xes. We want to show that if execution of a section ...egins in a state with the assertion at its head true, then when the execution leaves that section, the assertion at the exit must also be true. By taking an assertion ...t the end of each of these sections and using the ...emantics of the program statement above it, one can ...enerate an assertion which should have held before ...at statement if the assertions after it are to be guaranteed true. Working up the trees one then generates ...f the assertions in dashed-line boxes in Figure 2. Each ...ection will then preserve truth from its first to its last ...ssertions if the first assertion implies the assertion ...hat was generated in the dashed-line box at the top. ...ne thus gets the logical theorems or verification conditions given below each section. With a little thought it ...an now be seen that if these theorems can all be proven ...nd if the program halts, then it will halt with the ...orrect output values. In this case the theorems are ...bviously true. Halting can be proven by other techniques.

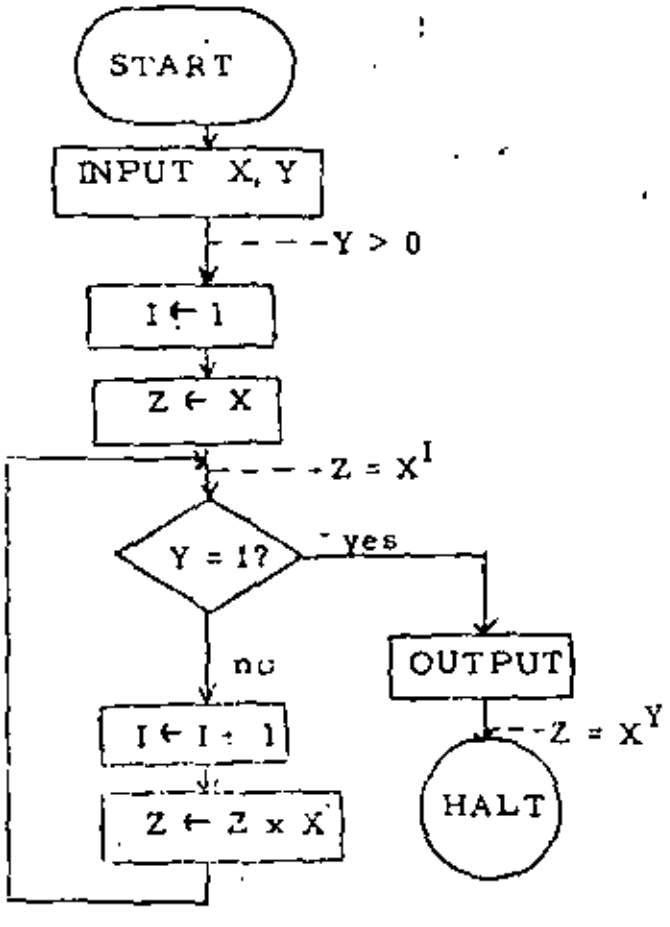
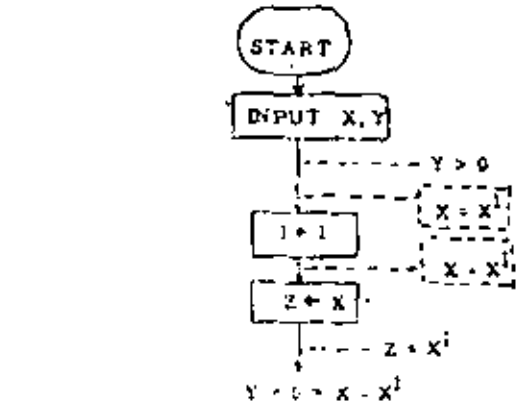
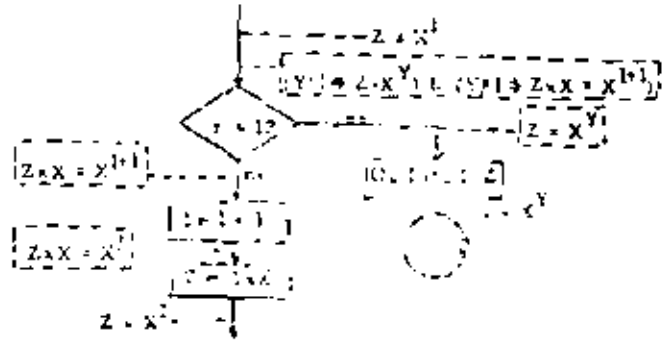


Figure 1—Exponentiation program





$Z = X^I \wedge [Y = I \Rightarrow Z = X^I \wedge (Y \neq I \Rightarrow Z \times X = X^{I+1})]$

...ed flowchart

The careful reader will note that the input assumption $Y > 0$ is not really needed for the proof of either of these theorems. This is because that assumption is really only needed to prove that the program terminates.

Inherent difficulties

This process for proving the correctness of programs is subject to many difficulties both to handle programming constructs which do not occur in this example and to try to make the proof of correctness more efficient. Full treatments with many examples are available in a recent article by King[2] and in Manna's forthcoming textbook.[3] Some further general comments about the nature of the problem will be made here. Analogous comments could be made about most of the other approaches to proving correctness.

Programs can only be said to be correct with respect to formal specifications or input-output assertions.

There is no formal way to guarantee that these specifications adequately express what someone really wants the program to do.

Given a program with specifications on the input and output, there is probably no automatic way to generate all the additional assertions which must be added to make the proof work. For a human to add these assertions requires a thorough understanding of the program. The programmer should be able to supply these assertions if he is able to formalize his intuitive understanding of the program.

Given a program with assertions in each loop and given an adequate definition of the semantics of the programming language, it is fairly routine to generate the theorems or verification conditions. Several existing computer programs that do this are described below.

The real problem in proving correctness lies in the fact that even for simple programs, the theorems that are generated become quite long. This length makes proving the theorems very difficult for a human or for current automatic theorem provers.

Formalizing the programmer's intuition of correctness

It may not be apparent, but the process of proving correctness is just a formalization into rigorous logical terms of the informal and sometimes sloppy reasoning that a programmer uses in constructing and checking his program. The programmer has some idea of what he expects to be true at each stage of his program (the assertions), he knows how the programming language semantics will transform a stage (generating the assertions in dashed-line boxes of Figure 2), and he convinces himself that the transformations will give the desired result (the proof of the theorem). In this sense proving program correctness is just a way to put into formal language everything one should understand in reading and informally checking a program for correctness. In fact, there is no clear division between the idea of reading code to check it for correctness and the idea of proving it correct by more rigorous means; the difference is one of degree of formality.

One question that should be addressed in this context regards the fact that both the correctness and the halting problems for arbitrary programs are known to be undecidable in the mathematical sense. However, this question of mathematical undecidability should not arise for any program for which there are valid intuitive reasons for the program to be correct.

Confidence in correctness

I hope I have made the point that logical proof of correctness techniques are radically different from testing techniques which are based on executing the program on selected input data in a specific environment. However, I do not want to imply that in a practical situation a proof or anything else can lead to absolute certitude of correctness. In fact a proof by itself does not necessarily lead to a higher level of confidence than might be achieved by extensive testing of a program. From a practical viewpoint there are a number of things that could still be wrong after a proof if one is not careful: what is proven may not be what one thought was proven, the proof may be incorrect, or assumptions about either the execution environment or the problem domain may not be valid. However, a proof does give a quite different and independent view of program correctness, and if it is done well, it should be able to provide a very high level of confidence in correctness. In particular, to the extent that a proof is valid, there should no longer be any doubt about what might happen after allowable but unexpected input values.

MANUAL PROOFS

The basic ideas in the last section have been known for some time. This section describes the practical progress which has been made with manual proofs in the last few years.

The size of programs which can be proven by hand depends on the level of formality that is used. In 1967 McCarthy and Painter[4] manually proved the correctness of a compiler for very elementary arithmetic expressions. It was a formal proof based on formal definitions of the syntax and semantics of the simple languages involved.

Rigorous but informal proofs

A more informal approach to proofs is now popular. This approach is rigorous, but uses a level of formality like that in a typical mathematics text. Arguments are based on an intuitive definition of the semantics of the programming language without a complete axiomatization. Using these techniques a variety of realistic, efficient programs to do sorting, merging, and searching have been proven correct. The proof of a twenty line sort program might require about three pages. It would now be a reasonable exercise for advanced graduate students.

Proofs of significantly more complex programs have also been published. London[6,7] has done proofs of a pair of LISP compilers. The larger compiler is about 160 lines of highly recursive code. It complies almost the

full LISP language—enough so it can compile itself. It is a generally unused compiler. It was written for teaching purposes, but it is not just a toy program. Another complex program has been proven correct by Jones.[8] The program is a PL-1 coding of a slightly simplified version of Earley's recognizer algorithm. It is about 200 lines of code. Probably the largest program that has been proven correct is in the work on computer interval arithmetic by Good and London.[9] There they proved the correctness of over 400 lines of Algol code. The largest individual procedure was in the 150-200 line category. A listing of many other significant programs which have been proven correct can be found in London's recent paper.[10]

If a complex 200 line program can now be proven correct by one man in a couple of months, one can begin to think about breaking larger programs into modules and getting a proof of correctness within a few man years of effort. Clearly there are programs for which a guarantee of the correctness of the running program would be worth not man years but many man decades of effort. We had better take a closer look at the feasibility of such an undertaking and what the proof of correctness would really accomplish.

Environment problems

In most existing proofs of program correctness, what has been proven correct is either the algorithm or a high level language representation of the algorithm. With today's computers what happens when the program actually runs on a physical computer would still be anybody's guess. It would be a significant additional chore to verify that the environment for the running program satisfies all the assumptions that were made about it in the proof. Problems with round off errors, overflow, and so forth can be handled in proofs. Good and London,[9] Hoare,[11] and others have described techniques for proving properties of programs in the context of computer arithmetic, but this can make the proof much more complex. Furthermore, to assure correctness of the running program one would have to be sure that all assumptions about the semantics of the programming language were actually valid in the implementation. The compiler and other system software would have to be certified. Finally, this could all be for naught considering the possibility of hardware failure as it exists in today's machines.

Thus, proving the correctness of a source language program is only one aspect of the whole problem of guaranteeing the correctness of a running program. Nevertheless, eliminating all errors from the source language program would certainly go a long way toward improving the probability that the program will run according to specifications.

Errors in the proof

An informal proof of correctness typically is much longer than the program text itself—often five to ten times as long. Thus the proof itself is subject to error just like any other extremely detailed and complex task done by humans. There is the possibility that an informal proof is just as wrong as the program. However, a proof does not have any loops and the meaning of a statement is fixed and not dependent on the current internal state of the computer. To read and check a proof is a straightforward and potentially automatable operation. The same can hardly be said for programs. Despite its potential fallibility, an informal proof would dramatically improve the probability that a program is correct. There is evidence from London's work[9] that a proof of correctness will find program bugs that have been overlooked in the code.

Less rigorous proofs

A person proving a program correct by manual techniques must first achieve a very thorough understanding of all details of the program. This clearly limits manual proof techniques to programs simple enough to be totally comprehended by the program provers. It also means that clarity and simplicity is very important in the program design if the program is to be proven correct. There is another school of thought which places primary emphasis on techniques for obtaining clarity and structure in the program design. Dijkstra[12,13] has long been the primary advocate of this approach. By appropriately structuring the program and by using what is apparently a much less formal approach to proofs, Dijkstra claims to have proven the correctness of his THE operating system.[14] Mills[15] advocates a similar approach with the program being sufficiently structured so an informal proof can be as short as the program text itself.

It is probably true that more practical results can be obtained with less rigorous approaches to proofs, especially in the near future. It is even debatable whether the more rigorous proofs give more assurance of correctness, but the formality does make it more feasible to automate the proof process. Whether or not one feels that the rigorous hand proofs of correctness will have much practical value, they are providing experience with different proof techniques that should

be very valuable in attempting to automate the proof process.

AUTOMATING PROOFS OF CORRECTNESS

In proving program correctness the logical statement that has to be proven usually is very long; however, the proof is seldom mathematically deep and much of it is likely to be quite simple. In the example given previously the theorems to be proven were almost trivial. It would seem that some sort of automatic theorem proving should be able to be applied in proving program correctness. This has been tried. So far the results have not been very exciting from a practical viewpoint.

Computer-generated proofs

Fully automatic theorem provers based on the resolution principle generally can prove correctness for very small programs—not much larger than the exponentiation program above. However, Slagle and Norton[20] report that they have obtained fully automatic proofs of the verification conditions for Hoare's sophisticated little program, FIND[5] which finds the kth largest element of an array. In 1969 King[21] completed a program verifier that automatically generated the verification conditions and then used a special theorem prover based on a natural deduction principle to automatically prove them. This system successfully proved programs to do a simple exchange sort, to test whether a number is prime, and similar integer manipulation programs. The data types were limited to integer variables and one dimensional arrays. Others have experimented with other data types and proof procedures. At the time of this writing I believe that there is no automatic theorem prover which has proven correctness for a program significantly larger than those mentioned.

Automatic theorem provers still cannot handle the straightforward complexity of the theorems that result from even fairly simple programs. Another problem lies in the fact that some of the properties of the programming language and, frequently, most of the application area of the program must be supplied to the theorem prover as axioms. Automatic theorem provers have difficulty in using axioms when they are given a large number of them. There is the minor success that have been reported as to what a tailor-made set of axioms for a particular problem should be like.

Computer-aided proofs

There are now several efforts directed toward providing computer assistance for proving correctness. This takes the form of systems to generate verification conditions and to do proof checking, formula simplification and editing, and semiautomatic or interactive theorem proving. Unfortunately at this time almost any automation of the proof process forces one into more detailed formalisms and reduces the size of the program that can be proven. This is because the logical size of the proof steps that can be taken in a partially automated proof system is still quite small. Presumably this is a temporary phenomenon. It seems reasonable to expect that we will soon see computer-aided verification systems which make use of some automatic theorem proving and can be used to prove correctness of programs somewhat larger than those that have been proven by hand.

Igarashi, London, and Luckham[22] are developing a system for proving programs written in PASCAL. The verification condition generator handles almost all the constructs of that language except for many of the data structures. Their approach is based on the work of Hoare.[5,7]

Elspas, Green, Levitt, and Waldinger[23] are developing a proof of correctness system based on the problem-solving language QA4.[24] It will use the goal-oriented, heuristic approach to theorem proving which is characteristic of that language.

Good and Ragland[25] have designed a simple language NUCLEUS with the idea that a verification system and a compiler for the language could be proven correct. Both the verification system and the compiler would be written in NUCLEUS and the proofs of correctness would be based on a formal definition of the language. The intent is that the language would then be able to be used to obtain other certified system software.

These three systems give a general idea of the current work going on. A proof-checking system will be described in the next section. Several other interesting systems have been implemented and basic information about them is readily available in London's recent paper.[26]

Language level

Proofs of correctness are currently far behind testing techniques in terms of the size and complexity of the programs that can be handled adequately. It is very much an open question whether automated proof techniques will ever be feasible as a commonly used alter-

native to testing a program. Many arguments pro and con are too subjective for adequate consideration here; however, a few comments are in order before one uses the rate of progress in the past as a basis for extrapolating into the future.

Proofs are based on sophisticated symbolic manipulations, and we are still at an early stage of gathering information about ways to automate them. Existing proof systems have been aimed mostly at testing the feasibility of techniques. Few if any have involved more than a couple man years of effort—many have been conceived on a scale appropriate for a Ph.D. dissertation. If and when a cost-effective system for proving correctness becomes feasible, it will certainly require a much larger implementation effort.

Proofs may be practical only in cases where a very high level of confidence is desired in specified aspects of program behavior. With computer-aided proofs one could hope to eliminate most of the sources of error that might remain after a manual proof. As exemplified by the work of Good and Ragland,[23] the verification system itself as well as compilers and other system software should be able to be certified. If the basic hardware/software is implemented with a system such as LOGOS[24] for computer-aided design of computer systems, then there should be a reasonable guarantee that the implemented computer system meets design specifications. With sufficient error-checking and redundancy, it should thus be possible to virtually eliminate the danger of either design or hardware malfunction errors. By the end of this decade these techniques may make it possible to obtain virtual certitude about a program's behavior in a running environment. There are many applications in areas such as real-time control, financial transactions, and computer privacy for which one would like to be able to achieve such a level of confidence.

SOME THEORETICAL FRONTIERS

Proofs of program correctness involve one in a seemingly exorbitant amount of formalism and detail. Some of this is inherent in the nature of the problem and will have to be handled by automation; however, the formalisms themselves often seem awkward. The long formulas and excessive detail may result partially because we have not yet found the best techniques and notation. Active theoretical research is developing many new techniques that could be used in proving correctness. Research in this area, usually called the mathematical theory of computation, has been active since McCarthy's[25,26] early papers on the subject. I feel that practical applications for proofs of correctness will develop slowly unless new techniques for proving correctness can significantly reduce the awkwardness of the formalisms required. This section will describe some of the current ideas being investigated. The topics chosen are those which seemed more directly related to techniques for facilitating proofs of correctness.

Induction techniques for loops and recursion

Proving correctness of programs would be comparatively simple if programs had no loops or recursion. However, some form of iteration or recursion is central to programming, and techniques for dealing with it effectively in proofs have been a subject of intensive study. All the techniques use some form of induction either explicitly or implicitly. The method of inductive assertions described previously handles loops in flowcharts by the addition of enough extra assertions to break every loop and then appeals to induction on the number of commands executed. For theoretical purposes it is often easier and more general to work with recursively defined functions rather than flowcharts. Almost ten years ago McCarthy proposed what he called Recursion Induction[26] for this situation. Manna et al. have extended the inductive assertion method to cover recursive,[27] parallel,[28] and non-deterministic[29] programs. Several other induction principles have been proposed by Burstall,[30] Park,[31] Morris,[32] and Scott.[33] A development and comparison of the various induction principles has been done recently by Manna, Ness, and Vuillemin.[34]

Formalizing the semantics of programming languages

The process of constructing the verification conditions or logical formulation of correctness is dependent on the meaning or semantics of the programming language. One can also take the opposite approach—proving correctness is a formal way of knowing whether a higher level meaning is true of the program. Thus the meaning or semantics of any program in a language is implicitly defined by a formal standard for deciding whether the program satisfies a specification. There is a very close interrelation between techniques for formalizing the semantics of a programming language and proofs of program correctness. Floyd's early work on assigning meanings to programs[1] has been developed especially by Manna[2] and Ashcroft.[35] Burstall[36] gives an alternative way to formulate program semantics in first-order logic. Ashcroft[37] has recently summarized this work and described its relevance.

Hoare,[21,22] Igarashi,[34] de Bakker,[35] and others have worked to develop axiomatic characterizations of the semantics of particular programming languages and constructs. The Vienna Definition Language[40] uses an abstract machine approach to defining semantics, and Allen[41] describes a way of obtaining an axiomatic definition from an abstract machine definition. The axiomatic definition is generally more useful in proofs. Scott and Strachey have developed another approach to defining semantics[42] which is described below.

Work on defining the semantics of programming languages is very active with many different approaches being tried. Those described above are only the ones more closely related to proofs. If any of these ideas can greatly simplify the expression or manipulation of properties of programs, they should have a similar simplifying impact on proofs of correctness.

Formal notation for specifications

Formal correctness only has meaning with respect to an independent, formal specification of what the program is supposed to do. For some programs such specifications can be given fairly easily. For example, consider a routine SORT which takes a vector X of arbitrary length n as an argument and produces a vector Y as its result. With appropriate conventions, the desired ordering on Y is specified by:

$$(\forall i,j)[1 \leq i < j \leq n \Rightarrow Y(i) \leq Y(j)]$$

One also needs a specification about the relation between X and Y. With the property $\mathrm{PERM}(x)$ meaning "x is a permutation" and using $\circ$ for functional composition, the following will do:

$$(\exists P)[\mathrm{PERM}(P) \& Y = X \circ P]$$

Note that the specification allows for any one of many possible algorithms to be chosen—presumably on the basis of efficiency. Yet from an external point of view the specification is complete. If SORT is to be used as part of a larger program, the specifications contain all one may want to know about it.

We can usually define correctness in this way for numeric, mathematical, and other simple programs typically found in program libraries. In fact the causality is largely the other way around: it is worth putting a program in a library to the extent that there is a good way of precisely defining the effects of the program without getting into all the details of its algorithmic implementation.

It would be useful to have a good way of writing formal specifications for a much wider range of computational processes. Parnas has been working on such techniques for formally specifying software modules.[43] His approach does handle error messages, and all side effects have to be carefully formalized.

From a proof of correctness point of view the formalism must have convenient deductive techniques as well as expressive power. First-order predicate calculus has the best deductive techniques, but without extensive definitions and axioms, its expressive power is very poor. For the SORT program above we assumed a definition of permutation, and still the specifications are more obscure than one might desire. For many programs the attempt to define their external effects with the formalism of a fairly standard predicate calculus can lead to extremely long and complex expressions. In particular, proof techniques associated with iteration and recursion have often been awkward when expressed in formal logic. One reason is that recursion and iteration lead to partial functions, that is, functions that may not be defined at all points. There has been a need for the logic that handles undefined values and can be easily used to prove properties of partial functions. Despite many efforts there has been no really successful, agreed-upon logical calculus that dealt with undefined values in a clean and natural way. Some recent work by Scott offers a possible solution to this and other problems.

The work of Scott, Strachey, and Milner

In 1964 Strachey[44] outlined an approach to defining the semantics of a programming language by mapping programs into a mathematical structure built up from a rather small number of precisely specified basic concepts. The approach eliminated any need for an abstract evaluating mechanism. Unfortunately the idea required some mathematical objects (such as self-referential functions) for which there was no firm mathematical foundation.

In 1969 Scott started to work on the underlying mathematical problems. The main breakthrough led to the first mathmeatical model of the λ-calculus.[45,46] The work involved the breaking of new ground in both lattice theory and topology. Function spaces are considered as lattices by using the "is consistent with and less defined than" relation on partial functions for the lattice partial ordering. It is then possible to define a logic with a fairly natural induction scheme which seems to have great generality and ease of expression for proving properties of recursively defined functions.

Scott's techniques allow for the construction of a universe of computable mathematical functions which

is sufficiently general so that it should be possible to define the meaning of any program by associating with it a specific function in this universe.[42,47] The semantics of a program are thus defined mathematically in terms of a limited number of basic mathematical concepts and not in terms of the result of a calculation on a machine. The semantical function that makes the association is defined recursively on the syntax of the program. The mathematical universe is sufficiently general so that the semantical function itself exists within the universe.[42]

The practicality of this approach has yet to be determined, but it seems to hold out the hope of a much less cumbersome way to formalize semantics. This mathematical approach to semantics may enable one to abstract from the arbitrary choices a great amount of extraneous detail that is typical of program implementations. The trick, of course, is to abstract from the right detail without losing important properties of the program.

Milner[48] has implemented a mechanical proof checker for a logic of computable functions based on some of the work of Scott. The implementation includes extensive simplification mechanisms and an interactive goal setting structure. Milner and Weyhrauch have used the logic to formalize semantics,[49,48] to prove simple program correctness,[50] and to give a mechanical proof of compiler correctness based on formally defined semantics.[51] The proof checker is still limited to proving properties of rather small programs; however, expressing formal properties of programs does seem to be simplified. The expression simplification mechanisms have also been useful.

The nature of this and other active theoretical research indicates that there may soon be techniques which will significantly simplify the problem of proving program correctness.

INTEGRATING PROOFS WITH PROGRAM DESIGN

Proving program correctness has usually been done after a program is written. An alternate approach is to integrate the proof with the program design. This approach provides some hope that proofs might eventually help to organize and simplify the program production process. A proof of correctness will greatly increase the amount of formalism that must be dealt with. However, if a proof can be integrated into the design and writing stages, it should eliminate most of the need for debugging and may alleviate the problems of documentation and maintenance. Floyd[52] has envisioned an automated verification system such that a programmer can interact with it in real time as he is writing his program.

Hoare's proof of correctness for his program FIND[39] was done in a top down way with the program and the proof evolving simultaneously. Jones in his proof of Earley's recognizer algorithm[3] exemplified a process he calls the formal development of correct algorithms. It is the longest published example of how a proof might discipline program design.

Throughout the development of the algorithm Jones uses a special formal notation related to the Vienna Definition Language, he does not introduce an ordinary programming language until the very end. With this notation he was able to give a formal, non-procedural specification for a recogniser in about half a page. He then develops the algorithm by stages while at each stage extending a proof that the partially developed algorithm will meet the specifications. At each stage the proof depends on formally expressed assumptions about the undeveloped part of the algorithm.

At the present time the amount of formalism required for the proof tends to overwhelm the program design effort. Nevertheless, this approach appears to make proofs of correctness somewhat more practical in an actual programming environment.

Automatic program synthesis

Rather than writing both specifications and a program, one might want to let the computer create the program and thus be responsible for its correctness. One technique for automatic program synthesis is closely related to techniques for proving correctness. One proves that there is an output satisfying the specifications and then extracts a program from this proof.[34,35] By using induction in the proof, it is possible to construct programs with loops. Manna and Waldinger have given several examples of this.[34]

While automatic program synthesis would be more useful than proving correctness, automatic synthesis requires a much more difficult proof. Since techniques for generating the required proofs are the major unsolved problem in this whole area, this form of automatic program synthesis is a more long-range goal than proofs of correctness.

CONCLUSION

Work on proving properties of programs has progressed to the point where one can argue whether there will soon be useful results. It is mostly a matter of what one means by "useful".

The software engineer who is worried about large programming projects will find current proof techniques hopelessly inadequate for all the large scale problems that are the center of his concern. Even for small modules he will probably find that test methods are more cost-effective than rigorous proofs. One should be able to obtain very great confidence in the correctness of a moderate-sized program if the level of talent and resources that would be necessary for a rigorous proof were devoted to reading and testing the program. Considering the time it normally takes for research results to work their way into practical applications, I would expect that it will be at least three or four years before this situation changes significantly.

Within the next three or four years, less rigorous techniques for structuring, understanding, and checking a program may become widely used. More rigorous proof techniques could be useful on small critical modules where adequate confidence cannot be achieved by other means. In this case it may be worth the additional cost of a proof to obtain an independent evaluation of correctness.

While most work on proving correctness has been for programs written in higher level languages, the most useful early applications may occur either for algorithms at the hardware or microcode level or for the calling structure at the highest level in the design of a large program. In both cases there is a high priority on correctness, and one would like to be assured of correctness long before testing becomes possible.

If current research on simplifying and automating the proof process can significantly reduce the difficulty of proving correctness, then in a few years proofs may be commonly used on small critical modules. Gradually the proof techniques could then be extended to larger programs so that they can be more useful in implementing very reliable systems. It is unlikely that proof techniques will be cost-effective for routine programs within this decade, but the potential is there for eventually revolutionizing the software marketplace.

REFERENCES

1 R W FLOYD
Assigning meanings to programs
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 19-32

2 Z MANNA
The correctness of programs
Journal of Computer and System Sciences Vol 3 No 2 May 1969 pp 119-127

3 B ELSPAS K N LEVITT R J WALDINGER A WAKSMAN
An assessment of techniques for proving program correctness
Computing Surveys Vol 4 No 2 June 1972

4 Z MANNA
Introduction to the mathematical theory of computation
McGraw Hill Book Co Inc to be published

5 J McCARTHY J PAINTER
Correctness of a compiler for arithmetic expressions
Proceedings of a Symposium in Applied Mathematics Vol 19 Mathematical Aspects of Computer Science American Mathematical Society 1967 pp 33-41

6 R L LONDON
Correctness of a compiler for a LISP subset
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 121-127

7 R LONDON
Correctness of two compilers for a LISP subset
Artificial Intelligence Memo 151 Stanford Univ Oct 1971

8 C B JONES
Formal development of correct algorithms: An example based on Earley's recogniser
Proceedings of an ACM Conference on Proving Assertions about Programs
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 Jan 1972 pp 150-169

9 D I GOOD R L LONDON
Computer interval arithmetic: Definition and proof of correct implementation
Journal of the ACM Vol 17 No 4 Oct 1970 pp 603-612

10 R L LONDON
The current state of proving programs correct
Proceedings of the ACM Annual Conf ACM 1972

11 C A R HOARE
An axiomatic basis of computer programming
Communications of the ACM Vol 12 No 10 Oct 1969 pp 576-583

12 E W DIJKSTRA
Notes on structured programming
Technische Hogeschool Eindhoven August 1969

13 E W DIJKSTRA
A constructive approach to the problem of program correctness
BIT Vol 8 1968 pp 174-186

14 E W DIJKSTRA
The structure of the "THE" multiprogramming system
Communications of the ACM Vol 11 No 5 May 1968 pp 341-346

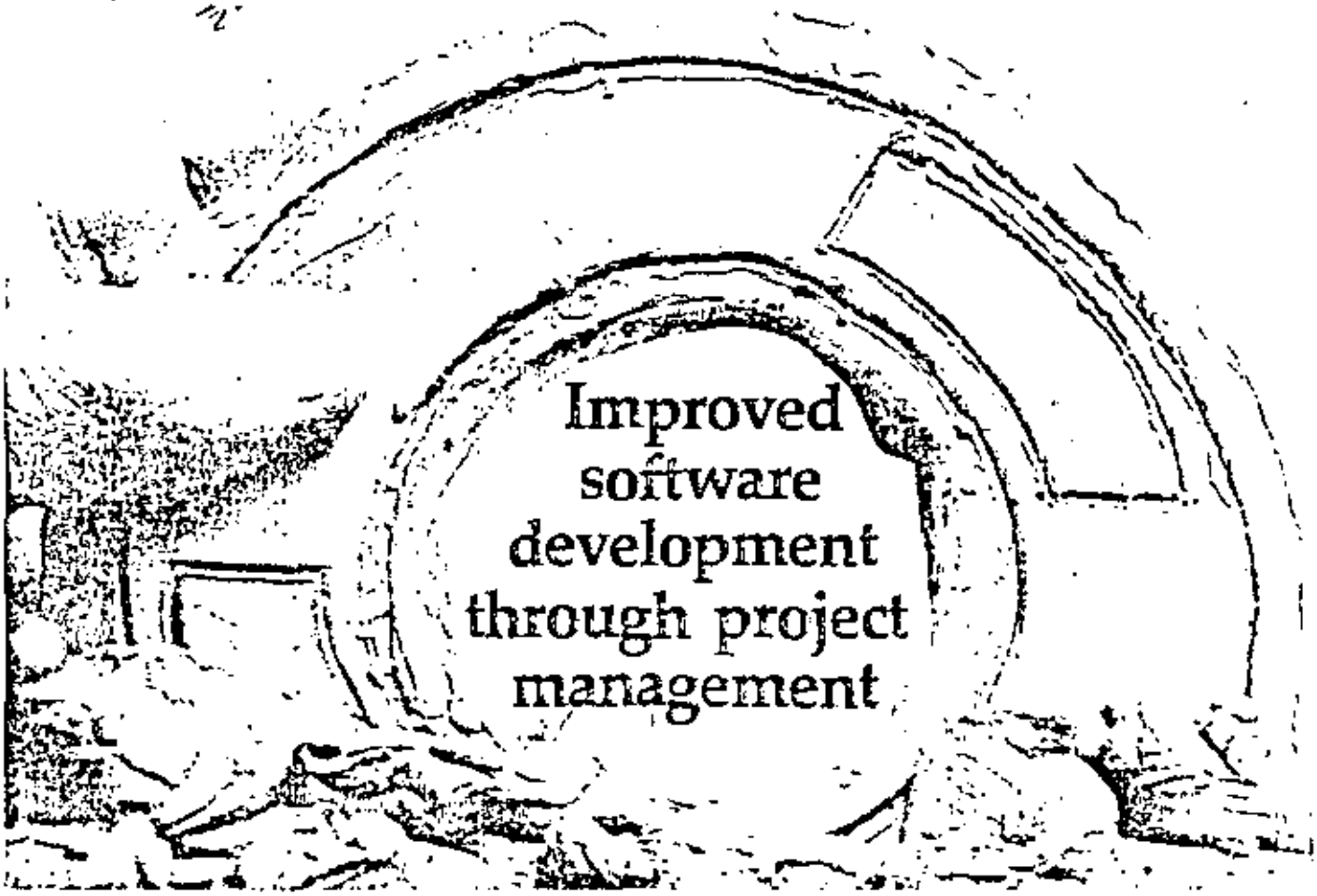
15 H D MILLS
The complexity of programs
Proc of SIGPLAN Symposium on Computer Program Test Methods Prentice-Hall to appear

16 J R SLAGLE L M NORTON
Experiments with an automatic prover having partial ordering rules
Heuristics Laboratory, National Institutes of Health 1971

17 C A R HOARE
Algorithm 65, find
Communications of the ACM Vol 4 No 7 July 1961 p 321

18 J C KING
A program verifier
PhD Thesis Carnegie-Mellon University Sept 1969

19 S IGARASHI R L LONDON D LUCKHAM
Private communication



Improved software development through project management

by Robert B. Fireworker and Leonard J. Bogner, Jr.

There is a need for improvement in state-of-the-art project management and programming techniques which aim to improve software development.

Customarily, programming projects begin with the user's problem statement and needs analysis. This in turn, is delivered to the software development group. Promises are made: Timely delivery (commitments, dates), accuracy, stated cost, system reliability and enhanced system functions. At this point, user/DP communication become secondary.

The typical product delivered:

- Functions—are they worth the cost and are they actually useful?
- Late installation—project slippage, poor estimates and/or change control.
- Resources—greater cost than anticipated.
- Questionable (interpretative) reliability.

A system containing these "features" must be further customized to meet the user's actual needs. Yes, an effective system may have been developed, with little help from original estimates; but how efficient and productive was the development process?

These items are symptomatic of software development problems.

The problems are poor planning (estimating, scheduling, control) and poor communication.

> *"We are still in the unfortunate condition that software development is not a science, it is a craft and our knowledge is of the meager and unsatisfactory kind."*—M. I. Bernstein "Hardware is Easy: It's Software That's Hard," DATAMATION.

Not to dissent the creative, artistic atmosphere of systems design, standardization must exist in terms of planning, documentation and review—in order to relieve ambiguity. Emphasis must not only be placed on what the user thinks he or she needs, but a determination should be made of present needs as well as needs for the future.

The elements of successful project management—planning, estimating and scheduling, tracking and control of change—will be the crux here. Modern programming technologies, as effective and efficient development tools, will be handled after that.

Software development cycle

The standard which evolved as the structure for development projects is the project life cycle (or software development cycle). Based on the premise that software development has birth (inception), maturity (development) and death (completion), the life cycle serves as a framework for communication and cooperation. It is utilized to monitor activities, highlight critical tasks and to record progress.

> *"The programming development cycle is simply a series of orderly, interrelated activities leading to the successful completion of a set of programs."*—Phillip W. Metzger, *Managing a Programming Project*, Prentice-Hall.

Standardization is evident through documentation of project phases, milestones and estimates. Milestones

are defined "end-products" which result from each phase. Estimates are judgments based on past experience and/or research. Ideally, estimates should be refined phase by phase. They should not be specific pinpoints of time and/or cost, but present an acceptable range.

By using the system life cycle as a tracking device, where estimates are compared to reality, project reviews should be conducted periodically. The number of reviews depends on the size and complexity of the project, experience of staff and events which occur. It is a good idea to plan a major milestone review after the completion of the conceptual design. This review should include all parties involved with emphasis given to user feedback concerning design direction.

Where are we now, and where are we going? As the project grows it becomes people intensive (See Chart A). If the design is off-course at the outset, it will result in timeframe slippage and increased costs further down the road.

Where the life cycle approach moves away from convention is in its flexible nature to accomplish its objectives.

Differences in user application are seen in phase titles, definitions, number and subdivisions. This flexibility allows the option of a broadly defined life cycle or a narrow, thorough breakdown. Chart B depicts various user interpretations of the life cycle. Although approaches to this technique differ, the objectives are the same—to document the project plan by task and timeframe.

The life cycle can be broken into five fundamental stages (See Chart A):

Planning
Design
Development
Testing
Implementation

Planning

Expectations are set during the initial stage of software development.

Productivity will be judged on how the appropriate expectations are met. Extensive time, up front, given to a well thought-out plan is invaluable to the success of a project. Metzger recommends 33 percent time allocation for planning process, while Dr. Raymond Winters (IBM) recommends 45 percent given up front. Coordination and utilization of all parties concerned during this process will stimulate commitment and lessen ambiguity.

Long term planning is the responsibility of upper management (both user and development) and is characterized by a few people involved and broad topic discussion. The planning process is continuous throughout the life cycle. As development progresses, more people provide planning input as the specifics of the system are discussed. Project management must be aware of and monitor short term plans and their relationship to long term goals.

User interaction will determine the scope of the project. Through the use of problem analysis, requirements study and feasibility techniques, project management will form a baseline. The manager must investigate the project environment: Past, present and future.

Past—systems and applications.

Present—resources (people and money), politics.

Future—corporate strategy, expansion plans.

Following this interaction, objectives are set, responsibilities assigned, schedules (resources and time) are developed.

Change control must be addressed —as user's needs change he or she must be aware of its effect on the scope of the project. If change is imperative, the plan may have to be revised and both the user and development management must sign off on the revision. Additional or shifting responsibilities will be decided at this time.

Measurement techniques also will be designated. The most useful technique will serve multiple functions: Planning, scheduling, controlling, communicating and simulating.

> *"What has emerged over the years is a technique that employs the common planning and scheduling procedures of PERT and CPM."*— C. W. Burrill and L. W. Ellsworth: *Modern Project Management* (unedited manuscript, used with permission of the publisher, Burrill-Ellsworth Assoc., Inc.).

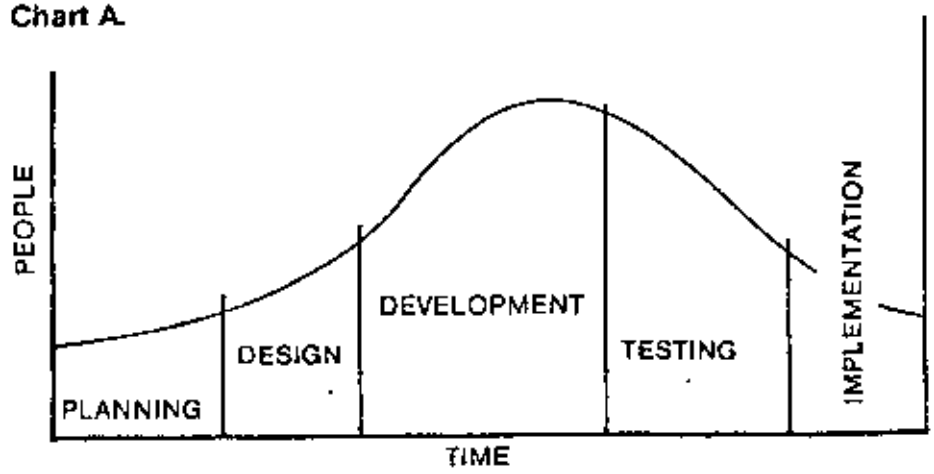
Networks (PERT/CPM) list the activities and their sequence within the life cycle. They provide a simple procedure for establishing a time schedule and monitor the critical activities involved. Through constant comparison of estimates to actual, project control is exercised. A documented network provides a communicative project plan. Networks also may be used for simulation as assessments of change and its impact may be observed affecting time and cost factors.

Top management's goals are overseeing long term planning.

Project management must also acknowledge:

- Temptation to design isolated from planning.
- Initial emphasis on priorities—not dates.
- Do not assume a "perfect world." Provide for contingencies such as illness, vacation and turnover.

Chart A.



Design

Once objectives are set and resources allocated, the next stage is to specify how the program system is to work.

The "conceptual design document" is the major outcome of the design phase. It serves as a blueprint for the design specifications (solutions to the problem) and serves as a starting point for the programmer.

The conceptual design document provides narrative portraying the overall concept. High-level explanations are given to these areas:

- Programming standards—covering flowcharting, data naming, interfacing. Prohibitions also should be stated.
- Program design—the actual structure of the system, as an overview of the hierarchy and not a breakdown of unit specifics.
- File design—accompanies the program design, defining system files to be utilized and accessed by program modules.
- Data flow—a narrative description of how the system will perform its tasks. This will be presented to non-technical management.

If possible, the conceptual design should define optional approaches and tradeoffs associated with each approach. Management will observe the long term and short term benefits of each and determine a course of action.

A major milestone review is to be conducted with the objective of confirming the design direction in terms of the user's expectations. This review will determine if modifications are to be made and to approve the continuation of the project.

Development

This is the most people intensive phase of the life cycle. It is the heart of the project because development is where the user's expectations will be met through products developed.

Development is people intensive: The final product must meet the user's expectations.

Because of the technical complexity of major programming projects, segregating the development functions into four groups is advised.

Four typical groups and their functions are as follows:

Programming group

Since the programmers are the focal point of the project, this section centers on their function.

From the conceptual design document, the programmers will develop a detailed design. The most current approach utilized is top-down design (discussed later). The programmer's job is to design module, in detail, concurring to the conventions of the conceptual design.

Next, program code must be developed. The detailed design may have to be changed but these changes should remain within the domain of the programmer as long as they do not affect the conceptual design.

Each module must be thoroughly tested prior to integration within the design hierarchy. Test drivers (programs written to simulate environmental conditions) are a useful testing aid. "It is management's responsibility to provide a good test environment—predictable computer time and smooth interfacing with the computer facility," says Metzger.

As modules are constructed and tested they must be documented. Programmers are responsible for this documentation. Changes to the original detailed design should be inserted within the document.

Once modules are tested and documented, they must be integrated within the design hierarchy. Observation of the new modules' effects on others and the system as a whole is the responsibility of programming management.

Analysis and design group

The analysts and designers remain active during the programming phase and serve the following functions:

- Change control—investigate, recommend, document.
- Data control—integrity of the system files.
- Review detailed design—compare to user expectations.
- User documentation—installation, operation, and maintenance.

Test group

This group prepares for and performs tests (see following information on testing) which are not concerned with programming but with overall systems tests. The separation of these functions allows programmers to concentrate only on code.

Staff support

Areas taken for granted are provided by this unit. Concerns include controlling computer time, supplying keypunch services, coordinating terminals and handling special fire-fighting assignments.

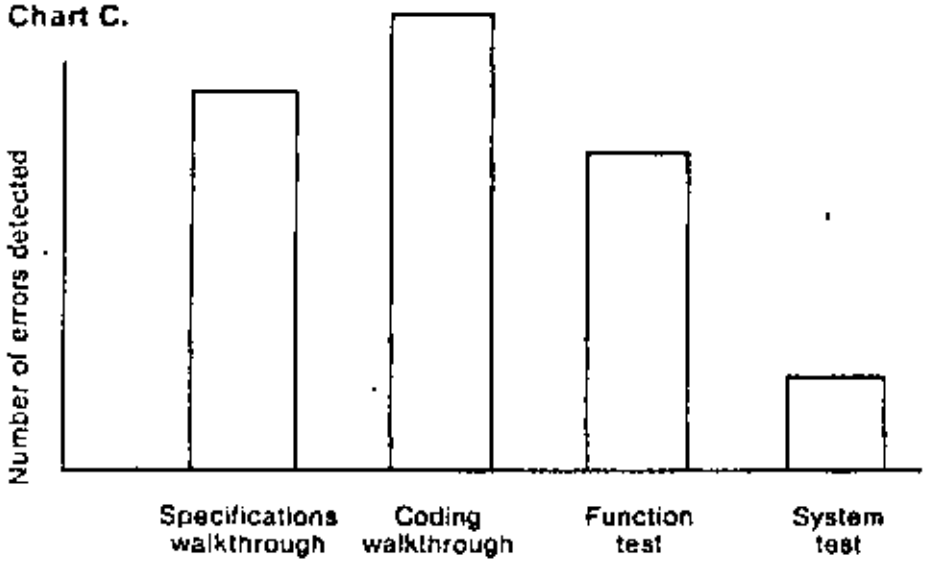
Testing

The main objective of the testing phase is to condition the programmers' products to an all-inclusive set of tests neither designed nor executed by the programmers and

Chart B.

Metzger	Montgomery Ward	Manufacturers Hanover
Definition	Planning	Feasibility
Design	Initial Investigation	Functional Analysis
Programming	Preliminary Study	Design
System Test	Systems Planning Study	Implementation
Acceptance Test	Development	
Installation & Operation	Systems Requirements	
	Systems Specifications	
	Technical Requirements	
	Implementation Planning	
	Programming	
	User Training	
	Systems Test	
	Implementation	
	Conversion	
	Post-Implementation Review	

Chart C.



run in as nearly a live atmosphere as possible. The secondary objective is to provide initial training to the user.

Responsibility for meeting these objectives belongs to the test group.

Upon receipt of the developed product and working with the aid of staff support, the tester must be prepared with specific and documented test procedures, scheduled computer time, library facilities (simulated and live data) and people ready to tear the system apart. The developed system should experience a thorough beating, for if it cannot handle almost any situation imaginable the system may not be accepted.

Observers are welcome, from both the user and programming side, to offer feedback and recommendations. It must be understood that this is a "test" and not a "demonstration" of the finalized system.

If it is determined that a program change must be made, a regression test should be performed. This is to discover the effect of the change on portions tested previously.

Documentation must contain listings of final test runs, programs and corresponding (narrative) documentation.

User training must involve those who maintain and operate the system. Users are to be provided with operating manuals and formal operational training. If a firm does not provide a regular education program, the test group should prepare formal classroom and/or on-the-job training.

Accompanying training methods, users who also will maintain the system must be provided with the detailed design document and the detailed code. Trouble-shooting manuals, listing peculiarities of the system, are also advised.

Implementation

Successful implementation is the result of careful planning during the previous phases. Very few objects reach this stage without being implemented. This is the time that user expectations are to be met, change is formally created within the organization and information processing begins. The user has the ultimate responsibility to verify the system's functional acceptability.

Criteria for system acceptance include:

- Integrity of the system's business functions, user manuals and procedures.
- Demonstration of the user operations group's ability to run the system in a live environment.
- A full scale live environment test using all resources available, executing all procedures and options —with a decision on the worthiness of the programs.

State-of-the-art programming techniques

Traditionally programming has been characterized by its flair for running into the same problems. To name a few: Inability to easily integrate modules, redundancies in functions, superfluous code and inability to detect errors up front. Modern programming techniques relieve the severity of these occurrences.

In this section, some of these techniques are discussed along with observations from three program development projects which utilized them. These projects are: Hartford Insurance Group; Commercial Auto Ratemaker System; Manufacturers Hanover Trust Co. (MHT); Wholesale Demand Deposit Accounting System; and Standard Oil of California, Chevron Program Development System.

Structured environment: Design and programming

Structured design is a collection of practices and procedures, chosen to complement one another, along with rules for applying them. "At MHT, a design methodology also includes management techniques, documentation procedures, tools to aid the designer, standards for specification that serve as the input to the design process (i.e., functional specifications), and the implementation process (i.e., design specifications)."—A. Block and K. Hamilton, "Programmer Productivity in a Structured Environment," INFOSYSTEMS.

Chief programmer is in charge of the program design and for reviewing code, integration and testing.

Advantages of this methodology, as seen through MHT, include:

- Consistency—variance is detected with higher certainty and at an earlier time.
- Modularity—one module for one function, ease of change control and maintenance.
- Documentation—more time and thought given to design than the traditional approach, resulting in greater design integrity.

"Structured programming is based on a mathematically proven structured theorem which states that any program can be written using only the three control logic structures."—"Installation Management/Improved Programming Technologies," IBM.

The chief programmer is responsible for the overall program design; writing mainline routines, critical code, OS interfaces; data definitions; defining modules and assigning to subordinates; and for specifying interfaces between modules. He or she also is responsible for reviewing code, overseeing all integration and testing, and reporting project status to management.

Advantages (Standard Oil) are:

- Technical expertise at management level, reviewing and optimizing code.
- Expert handling critical code and delegating simpler routines to less experience (level of competence).

HIPO diagrams

...erarchy plus Input Process-Output (HIPO) diagrams are a documentation technique utilized during the design stage prior to the actual coding. "HIPO reduces the amount and ambiguity of the prose required to document function and

Major milestone review sizes up progress on the project: Is this what the user wants?

provides a systematic means of identifying all the functions to be performed and the modules to perform them."—"Installation Manag at," IBM.

HIPO renders a top-down description of program functions. First is an overview, followed by the details of the particular function and its necessary inputs and resultant outputs.

Typically, a HIPO package contains:

- A hierarchy chart that identifies all overview and detail diagrams within the top-down structure.
- Overview and detailed graphic descriptions: Input—files, records, fields, control blocks; process—interactions; output—files, records, control blocks.

Structured walkthroughs

This is an objective check on a program's overall logic and completeness by someone who has not been immersed in its details. "The structured walkthrough is designed to detect and remove errors as early as possible (See Chart C) during the cycle in a problem solving and non-fault finding atmosphere where everyone involved—especially the developer—is eager to find any errors in the work product being reviewed."—"Installation Management," IBM.

These structures are:

- Sequence of two or more operations (MOVE, ADD, . . .).
- Conditional branching to one of two operations and return (IF THEN ELSE).
- Repetition of an operation while a condition is true (DO WHILE).

This method is characterized by the absence of GO TOs and have only one entry and one exit. Since arbitrary branching is not utilized, modules of code are easy to read. Flexibility is allowed through extensions to logic as long as the code retains its one-entry/one-exit property.

Advantages of structured programming, noted by the Hartford Insurance Group, are:

- Compact, accurate, easily followed code.
- Consistently fewer lines of code.
- Use of comments became almost unnecessary.
- Programmers found it easier to locate bugs in their code during tests (See Chart C).

Top-down design and program development

Top-down strategy is a hierarchical approach to program design and development. Similar to the overall systems plan, the initial design is to first design and code high-level modules followed by determining the low-level units which will be needed. "Testing of high-level modules is through the use of test subroutines and procedure calls."—"Improved Productivity in Implementing Information Systems," INFO 79 speech by Harold Feinlieb of National CSS, Inc., and Kevin Tweedy, Standard Oil of California.

Advantages to this method, as experienced by Standard Oil, are:

- Eliminates unnecessary code.
- Ease of integration.
- Effective management control through visible high level products.

Chief programmer teams

Here a small group of programmers (3-5) are under the direction of a senior level or chief programmer. The team represents an opportunity to improve both the manageability and productivity of the group through organizational techniques. These techniques include: Specialized programming jobs, expert technical leadership, stress on training and career development, defined relationships among specialists and effective work effort with a developing, and always visible, project.

The structured walkthrough is a review session during which the developer invites peers to observe his or her logic, step-by-step, and to offer constructive criticism. This technique has evolved not only as an error detection device but also as a communications tool, through discussion of personal approaches to program logic.

Advantages as viewed by the Hartford Group are illustrated in Chart C. ■

About the authors

Fireworker is professor of quantitative analysis in the graduate school of business, St. John's University, Jamaica, NY. He also directs the university's computer information systems academic program. With an ongoing consulting practice, Fireworker has done work for 3M Co., Lever Brothers Co., IBM, Consolidated Edison, Purolator and other large firms. He holds a Ph.D. in operations research and an MS in computer science from New York University. Bogner is a systems analyst with Manufacturers Hanover Trust Co. and is in charge of information and administrative services. He earned his MBA at St. John's University.—Ed.

References

Books

Burnill, C. W., and Ellsworth, L. W.: Modern Project Management, (unedited manuscript, used with permission of the publisher, Burnill-Ellsworth Assoc., Inc.) obtained at IBM Systems Science Institute.

Metzger, P. W. Managing a Programming Project, Prentice-Hall, Englewood Cliffs, NJ. 1973.

Technical Publications

IBM, Installation Management Series, "Managing Systems Development—Montgomery Ward," November, 1973, "Improved Pro...

Continued on page 35

Considerations and techniques are proposed that reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple, independent, reusable modules. Debugging and modifying programs, reconfiguring I/O devices, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

Structured design

by W. P. Stevens, G. J. Myers, and L. L. Constantine

Structured design is a set of proposed general program design considerations and techniques for making coding, debugging, and modification easier, faster, and less expensive by reducing complexity.[1] The major ideas are the result of nearly ten years of research by Mr. Constantine.[2] His results are presented here, but the authors do not intend to present the theory and derivation of the results in this paper. These ideas have been called *composite design* by Mr. Myers.[3,4] The authors believe these program *design* techniques are compatible with, and enhance, the *documentation* techniques of HIPO[5] and the *coding* techniques of structured programming.[7]

These cost-saving techniques always need to be balanced with other constraints on the system. But the ability to produce simple, changeable programs will become increasingly important as the cost of the programmer's time continues to rise.

General considerations of structured design

Simplicity is the primary measurement recommended for evaluating alternative designs relative to reduced debugging and modification time. Simplicity can be enhanced by dividing the system into separate pieces in such a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system. Observability (the ability to easily perceive how and why actions occur) is another useful consideration that can help in designing programs that can be changed easily. Consideration of the effect of reasonable changes is also valuable for evaluating alternative designs.

Mr. Constantine has observed that programs that were the easiest to implement and change were those composed of simple, independent modules. The reason for this is that problem solving is faster and easier when the problem can be subdivided into pieces which can be considered separately. Problem solving is hardest when all aspects of the problem must be considered simultaneously.

The term *module* is used to refer to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names. Examples of modules are PL/I procedures, FORTRAN mainlines and subprograms, and, in general, subroutines of all types. Considerations are always with relation to the program statements as *coded*, since it is the programmer's ability to understand and change the *source* program that is under consideration.

Figure 1 A structured chart

While conceptually it is useful to discuss dividing whole programs into smaller pieces, the techniques presented here are for designing simple, independent modules originally. It turns out to be difficult to divide an existing program into separate pieces without increasing the complexity because of the amount of overlapped code and other interrelationships that usually exist.

Graphical notation is a useful tool for structured design. Figure 1 illustrates a notation called a *structure chart*,[6] in which:

1. There are two modules, A and B.
2. Module A *invokes* module B. B is *subordinate* to A.
3. B receives an input parameter *X* (its name in module A) and returns a parameter *Y* (its name in module A). (It is useful to distinguish which calling parameters represent data passed *to* the called program and which are for data to be *returned* to the caller.)

Coupling and communication

To evaluate alternatives for dividing programs into modules, it becomes useful to examine and evaluate types of "connections" between modules. A connection is a reference to some label or address defined (or also defined) elsewhere.

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other

Table 1 Contributing factors

	Interface complexity	Type of connection	Type of communication
low	simple, obvious	to module by name	data
COUPLING			control
high	complicated, obscure	to internal elements	hybrid

modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous "ripple" effects, where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc. The widely used technique of using common data areas (or global variables or modules without their own distinct set of variable names) can result in an enormous number of connections between the modules of a program. The complexity of a system is affected not only by the number of connections but by the degree to which each connection couples (associates) two modules, making them interdependent rather than independent. Coupling is the measure of the strength of associatidn established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.

The degree of coupling established by a particular connection is a function of several factors, and thus it is difficult to establish a simple index of coupling. Coupling depends (1) on how complicated the connection is, (2) on whether the connection refers to the module itself or something inside it, and (3) on what is being sent or received.

Coupling increases with increasing complexity or obscurity of the interface. Coupling is lower when the connection is to the normal module interface than when the connection is to an internal component. Coupling is lower with data connections than with control connections, which are in turn lower than hybrid connections (modification of one module's code by another module). The contribution of all these factors is summarized in Table 1.

Interface complexity

When two or more modules interface with the same area of storage, data region, or device, they share a common environment. Examples of common environments are:

- A set of data elements with the EXTERNAL attribute that is copied into PL/I modules via an INCLUDE statement or that is found listed in each of a number of modules.
- Data elements defined in COMMON statements in FORTRAN modules.
- A centrally located "control block" or set of control blocks.
- A common overlay region of memory.
- Global variable names defined over an entire program or section.

The most important structural characteristic of a common environment is that it couples every module sharing it to every other such module without regard to their functional relationship or its absence. For example, only the two modules XVECTOR and VELOC might actually make use of data element X in an "included" common environment of PL/I, yet changing the length of X impacts *every* module making any use of the common environment, and thus necessitates recompilation.

Every element in the common environment, whether used by particular modules or not, constitutes a separate path along which errors and changes can propagate. Each element in the common environment adds to the complexity of the total system to be comprehended by an amount representing all possible pairs of modules sharing that environment. Changes to, and new uses of, the common area potentially impact all modules in unpredictable ways. Data references may become unplanned, uncontrolled, and even unknown.

A module interfacing with a common environment for some of its input or output data is, on the average, more difficult to use in varying contexts or from a variety of places or in different programs than is a module with communication restricted to parameters in calling sequences. It is somewhat clumsier to establish a new and unique data context on each call of a module when data passage is via a common environment. Without analysis of the entire set of sharing modules or careful saving and restoration of values, a new use is likely to interfere with other uses of the common environment and propagate errors into other modules. As to future growth of a given system, once the commitment is made to communication via a common environment, any new module will have to be plugged into the common environment, compounding the total complexity even more. On this point, Belady and Lehman,[8] observe that "a well-structured system, one in which communication is via passed parameters through defined interfaces, is likely to be more growable and require less effort to maintain than one making extensive use of global or shared variables."

The impact of common environments on system complexity may be quantified. Among M objects there are $M(M-1)$ or-

dered pairs of objects. (Ordered pairs are of interest because A and B sharing a common environment complicates both, A being coupled to B and B being coupled to A.) Thus a common environment of N elements shared by M modules results in $NM(M-1)$ first order (one level) relationships or paths along which changes and errors can propagate. This means 150 such paths in a FORTRAN program of only three modules sharing the COMMON area with just 25 variables in it.

It is possible to minimize these disadvantages of common environments by limiting access to the smallest possible subset of modules. If the total set of potentially shared elements is subdivided into groups, all of which are *required* by some subset of modules, then both the size of each common environment and the scope of modules among which it is shared is reduced. Using "named" rather than "blank" COMMON in FORTRAN is one means of accomplishing this end.

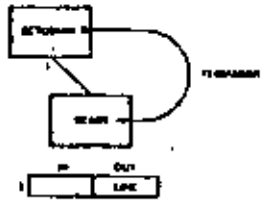
The complexity of an interface is a matter of how much information is needed to state or to understand the connection. Thus, obvious relationships result in lower coupling than obscure or inferred ones. The more syntactic units (such as parameters) in the statement of a connection, the higher the coupling. Thus, extraneous elements irrelevant to the programmer's and the modules' immediate task increase coupling unnecessarily.

Connections that address or refer to a module as a whole by its name (leaving its contents unknown and irrelevant) yield lower coupling than connections referring to the internal elements of another module. In the latter case, as for example the use of a variable by direct reference from within some other module, the entire content of that module may have to be taken into account to correct an error or make a change so that it does not make an impact in some unexpected way. Modules that can be used easily without knowing anything about their insides make for simpler systems.

type of connection

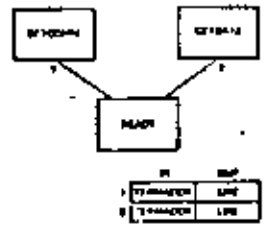
Consider the case depicted in Figure 2. GETCOMM is a module whose function is getting the next command from a terminal. In performing this function, GETCOMM calls the module READT, whose function is to read a line from the terminal. READT requires the address of the terminal. It gets this via an externally declared data element in GETCOMM, called TERMADDR. READT passes the line back to GETCOMM as an argument called LINE. Note the arrow extending from *inside* GETCOMM to *inside* READT. An arrow of this type is the notation for references to internal data elements of another module.

Figure 2 Module connections



Now, suppose we wish to add a module called GETDATA, whose function is to get the next data line (i.e., not a command) from a

Figure 3 Improved module connections

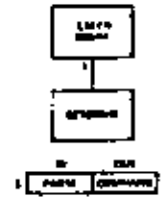


type of communication

(possibly) different terminal. It would be desirable to use module READT as a subroutine of GETDATA. But if GETDATA modifies TERMADDR in GETCOMM before calling READT, it will cause GETCOMM to fail since it will "get" from the wrong terminal. Even if GETDATA restores TERMADDR after use, the error can still occur if GETDATA and GETCOMM can ever be invoked "simultaneously" in a multiprogramming environment. READT would have been more usable if TERMADDR had been made an input argument to READT instead of an externally declared data item as shown in Figure 3. This simple example shows how references to internal elements of other modules can have an adverse effect on program modification, both in terms of cost and potential bugs.

Modules must at least pass data or they cannot functionally be a part of a single system. Thus connections that pass data are a necessary minimum. (Not so the communication of control. In principle, the presence or absence of requisite input data is sufficient to define the circumstances under which a module should be activated, that is, receive control. Thus the explicit passing of control by one module to another constitutes an additional, theoretically inessential form of coupling. In practice, systems that are *purely* data-coupled require special language and operating system support but have numerous attractions, not the least of which is they can be fundamentally simpler than any equivalent system with control coupling.[9])

Figure 4 Control-coupled modules



Beyond the practical, innocuous, minimum control coupling of normal subroutine calls is the practice of passing an "element of control" such as a switch, flag, or signal from one module to another. Such a connection affects the execution of another module and not merely the data it performs its task upon by involving one module in the internal processing of some other module. Control arguments are an additional complication to the essential data arguments required for performance of some task, and an alternative structure that eliminates the complication always exists.

Consider the modules in Figure 4 that are control-coupled by the switch PARSE through which EXECNCOMM instructs GETCOMM whether to return a parsed or unparsed command. Separating the two distinct functions of GETCOMM results in a structure that is simpler as shown in Figure 5.

The new EXECNCOMM is no more complicated; where once it set a switch and called, now it has two alternate calls. The sum of GETPCOMM and GETUCOMM is (functionally) less complicated than GETCOMM was (by the amount of the switch testing). And the two small modules are likely to be easier to comprehend than the one large one. Admittedly, the immediate gains here

may appear marginal, but they rise with time and the number of alternatives in the switch and the number of levels over which it is passed. Control coupling, where a called module "tells" its caller what to do, is a more severe form of coupling.

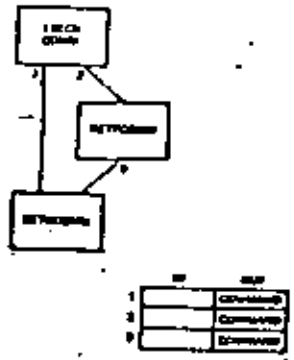
Modification of one module's code by another module may be thought of as a hybrid of data and control elements since the code is dealt with as data by the modifying module, while it acts as control to the modified module. The target module is very dependent in its behavior on the modifying module, and the latter is intimately involved in the other's internal functioning.

Cohesiveness

Coupling is reduced when the relationships among elements *not* in the same module are minimized. There are two ways of achieving this—minimizing the relationships among modules and maximizing relationships among elements in the same module. In practice, both ways are used.

The second method is the subject of this section. "Element" in this sense means any form of a "piece" of the module, such as a statement, a segment, or a "subfunction". Binding is the measure of the cohesiveness of a module. The objective here is to reduce coupling by striving for high binding. The scale of cohesiveness, from lowest to highest, follows:

1. Coincidental.
2. Logical.
3. Temporal.
4. Communicational.
5. Sequential.
6. Functional.



The scale is not linear. Functional binding is much stronger than all the rest, and the first two are much weaker than all the rest. Also, higher-level binding classifications often include all the characteristics of one or more classifications below it *plus* additional relationships. The binding between two elements is the highest classification that applies. We will define each type of binding, give an example, and try to indicate why it is found at its particular position on the scale.

coincidental binding

When there is no meaningful relationship among the elements in a module, we have coincidental binding. Coincidental binding might result from either of the following situations: (1) An existing program is "modularized" by splitting it apart into modules. (2) Modules are created to consolidate "duplicate coding" in other modules.

As an example of the difficulty that can result from coincidental binding, suppose the following sequence of instructions appeared several times in a module or in several modules and was put into a separate module called X:

```
A = B + C
GET CARD
PUT OUTPUT
IF B = 4 THEN F = 0
```

Module X would probably be coincidentally bound since these four instructions have no apparent relationships among one another. Suppose in the future we have a need in one of the modules originally containing these instructions to say GET TAPE RECORD instead of GET CARD. We now have a problem. If we *modify* the instruction in module X, it is unusable to all of the other callers of X. It may even be difficult to *find* all of the other callers of X in order to make any other compatible change.

It is only fair to admit that, independent of a module's cohesiveness, there are instances when any module can be modified in such a fashion to make it unusable to all its callers. However, the *probability* of this happening is very high if the module is coincidentally bound.

logical binding

Logical binding, next on the scale, implies some logical relationship between the elements of a module. Examples are a module that performs all input and output operations for the program or a module that edits all data.

The logically bound, EDIT ALL DATA module is often implemented as follows. Assume the data elements to be edited are master file records, updates, deletions, and additions. Parameters passed to the module would include the data and a special parameter indicating the type of data. The first instruction in the module is probably a four-way branch, going to four sections of code—edit master record, edit update record, edit addition record, and edit deletion record.

Often, these four functions are also intertwined in some way in the module. If the deletion record changes and requires a change to the edit deletion record function, we will have a problem if this function is intertwined with the other three. If the edits are truly independent, then the system could be simplified by putting each edit in a separate module and eliminating the need to decide which edit to do for each execution. In short, logical binding usually results in tricky or shared code, which is difficult to modify, and in the passing of unnecessary parameters.

94

Temporal binding is the same as logical binding, except the elements are also related in time. That is, the temporally bound elements are executed in the same time period.

temporal binding

The best examples of modules in this class are the traditional "initialization", "termination", "housekeeping", and "clean-up" modules. Elements in an initialization module are logically bound because initialization represents a logical class of functions. In addition, these elements are related in time (i.e., at initialization time).

Modules with temporal binding tend to exhibit the disadvantages of logically bound modules. However, temporally bound modules are higher on the scale since they tend to be simpler for the reason that *all* of the elements are executable at one time (i.e., no parameters and logic to determine which element to execute).

A module with communicational binding has elements that are related by a reference to the same set of input and/or output data. For example, "print and punch the output file" is communicationally bound. Communicational binding is higher on the scale than temporal binding since the elements in a module with communicational binding have the stronger "bond" of referring to the same data.

communicational binding

When the output data from an element is the input for the next element, the module is sequentially bound. Sequential binding can result from flowcharting the problem to be solved and then defining modules to represent one or more blocks in the flowchart. For example, "read next transaction and update master file" is sequentially bound.

sequential binding

Sequential binding, although high on the scale because of a close relationship to the problem structure, is still far from the maximum—functional binding. The reason is that the procedural processes in a program are usually distinct from the *functions* in a program. Hence, a sequentially bound module can contain several functions or just part of a function. This usually results in higher coupling and modules that are less likely to be usable from other parts of the system.

Functional binding is the strongest type of binding. In a functionally bound module, all of the elements are related to the performance of a single function.

functional binding

A question that often arises at this point is what is a function? In mathematics, $Y = F(X)$ is read "Y is a function F of X." The function F defines a transformation or mapping of the independent (or input) variable X into the dependent (or return) variable Y. Hence, a function describes a transformation from some

95

input data to some return data. In terms of programming, we broaden this definition to allow functions with no input data and functions with no return data.

In practice, the above definition does not clearly describe a functionally bound module. One hint is that if the elements of the module all contribute to accomplishing a single goal, then it is probably functionally bound. Examples of functionally bound modules are "Compute Square Root" (input and return parameters) "Obtain Random Number" (no input parameter), and "Write Record to Output File" (no return parameter).

A useful technique in determining whether a module is functionally bound is writing a sentence describing the function (purpose) of the module, and then examining the sentence. The following tests can be made:

1. If the sentence *has* to be a compound sentence, contain a comma, or contain more than one verb, the module is probably performing more than one function: therefore, it probably has sequential or communicational binding.

2. If the sentence contains words relating to time, such as "first", "next", "then", "after", "when", "start", etc., then the module probably has sequential or temporal binding.

3. If the predicate of the sentence doesn't contain a single specific object following the verb, the module is probably logically bound. For example, Edit All Data has logical binding; Edit Source Statement may have functional binding.

4. Words such as "initialize", "clean-up", etc. imply temporal binding.

Functionally bound modules can always be described by way of their elements using a compound sentence. But if the above language is unavoidable while still completely describing the module's function, then the module is probably not functionally bound.

One unresolved problem is deciding how far to divide functionally bound subfunctions. The division has probably gone far enough if each module contains no subset of elements that could be useful alone, and if each module is small enough that its entire implementation can be grasped all at once, i.e., seldom longer than one or two pages of source code.

Observe that a module can include more than one type of binding. The binding between two elements is the highest that can be

applied. The binding of a module is lowered by every element pair that does not exhibit functional binding.

Predictable modules

A predictable, or well-behaved, module is one that, when given the identical inputs, operates identically each time it is called. Also, a well-behaved module operates independently of its environment.

To show that dependable (free from errors) modules can still be unpredictable, consider an oscillator module that returns zero and one alternately and dependably when it is called. It might be used to facilitate double buffering. Should it have multiple users, each would be required to call it an even number of times before relinquishing control. Should any of the users have an error that prevented an even number of calls, all other users will fail. The operation of the module given the same inputs is not constant, resulting in the module not being predictable even though error-free. Modules that keep track of their own state are usually not predictable, even when error-free.

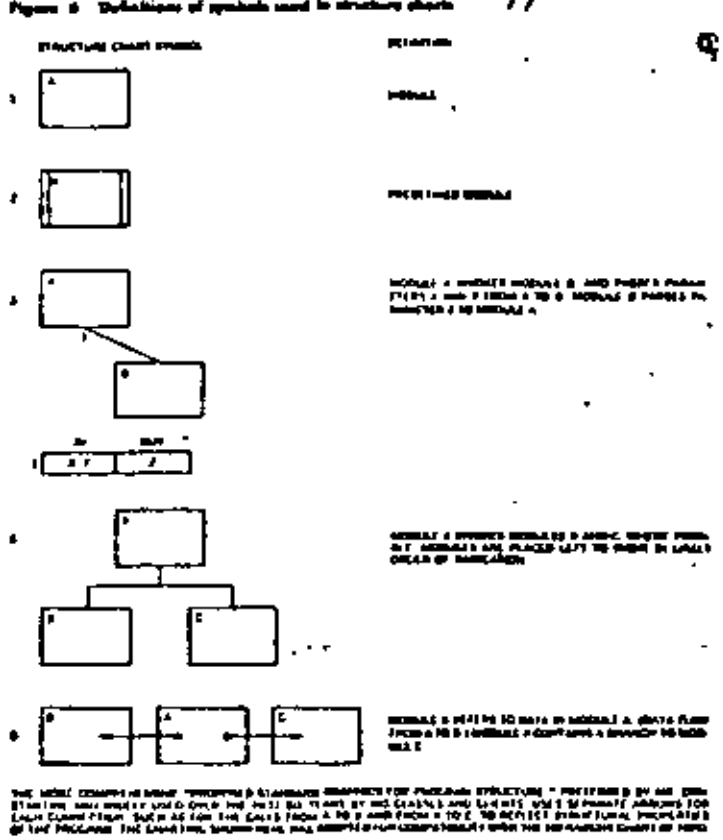
This characteristic of predictability that can be designed into modules is what we might loosely call "black-boxness." That is, the user can understand what the module does and use it without knowing what is inside it. Module "black-boxness" can even be enhanced by merely adding comments that make the module's function and use clear. Also, a descriptive name and a well-defined and visible interface enhances a module's usability and thus makes it more of a black box.

Tradeoffs to structured design

The overhead involved in writing many simple modules is in the execution time and memory space used by a particular language to effect the call. The designer should realize the adverse effect on maintenance and debugging that may result from striving just for minimum execution time and/or memory. He should also remember that programmer cost, is, or is rapidly becoming, the major cost of a programming system and that much of the maintenance will be in the future when the trend will be even more prominent. However, depending on the actual overhead of the language being used, it is very possible that a structured design can result in less execution and/or memory overhead rather than more due to the following considerations:

For memory overhead

1. Optional (error) modules may never be called into memory.



2. Structured design reduces duplicate code and the coding necessary for implementing control switches, thus reducing the amount of programmer-generated code.
3. Overlay structuring can be based on actual operating characteristics obtained by running and observing the program.
4. Having many single-function modules allows more flexible, and precise, grouping, possibly resulting in less memory needed at any one time under overlay or virtual storage constraints.

For execution overhead

1. Some modules may only execute a few times.
2. Optional (error) functions may never be called, resulting in zero overhead.
3. Code for control switches is reduced or eliminated, reducing the total amount of code to be executed.

4. Heavily used linkage can be recompiled and calls replaced by branches.
5. "Includes" or "performs" can be used in place of calls. (However, the complexity of the system will increase by at least the extra consideration necessary to prevent duplicating data names and by the difficulty of creating the equivalent of call parameters for a well-defined interface.)
6. One way to get fast execution is to determine which parts of the system will be most used so all optimizing time can be spent on those parts. Implementing an initially structured design allows the testing of a working program for those critical modules (and yields a working program prior to any time spent optimizing). Those modules can then be optimized separately and reintegrated without introducing multitudes of errors into the rest of the program.

Structured design techniques

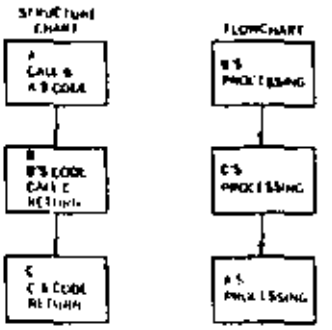
It is possible to divide the design process into general program design and detailed design as follows. General program design is deciding *what* functions are needed for the program (or programming system). Detailed design is *how* to implement the functions. The considerations above and techniques that follow result in an identification of the functions, calling parameters, and the call relationships for a structure of functionally bound, simply connected modules. The information thus generated makes it easier for each module to then be separately designed, implemented, and tested.

structure charts

The objective of general program design is to determine what functions, calling parameters, and call relationships are needed. Since flowcharts depict *when* in what order and under what conditions) blocks are executed, flowcharts unnecessarily complicate the general program design phase. A more useful notation is the structure chart, as described earlier and as shown in Figure 6.

To contrast a structure chart and a flowchart, consider the following for the same three modules in Figure 7 – A which calls B which calls C. Coding has been added to the structure chart to enable the proper flowchart to be determined: B's code will be executed first, then C's, then A's (!). To design A's interfaces properly, it is necessary to know that A is responsible for invoking B, but this is hard to determine from the flowchart. In addition, the structure chart can show the module connections and calling parameters that are central to the consideration and techniques being presented here.

Figure 7 Structure chart compared to flowchart



The other major difference that drastically simplifies the nota-

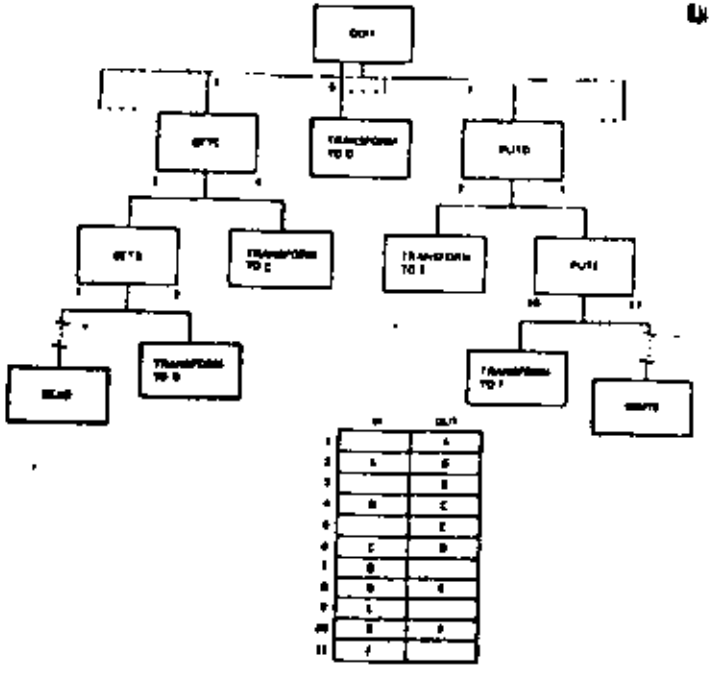
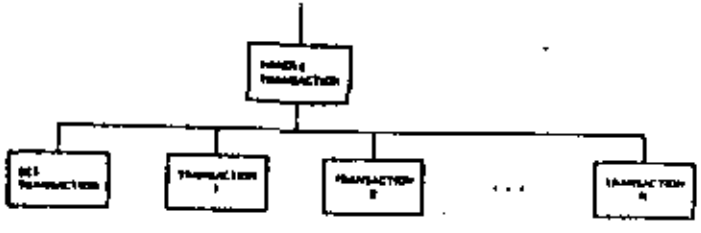


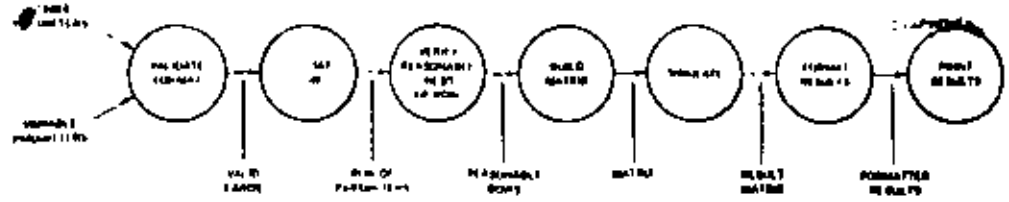
Figure 9 Transaction structure



tion and analysis during general program design is the absence in structure charts of the decision block. Conditional calls can be so noted, but "decision designing" can be deferred until detailed module design. This is an example of where the *design* process is made simpler by having to consider only part of the design problem. Structure charts are also small enough to be worked on all at once by the designers, helping to prevent suboptimizing parts of the program at the expense of the entire problem.

common structures

A shortcut for arriving at simple structures is to know the general form of the result. Mr. Constantine observed that programs of the general structure in Figure 8 resulted in the lowest-cost



implementations. It implements the input-process-output type of program, which applies to most programs, even if the "input" or "output" is to secondary storage or to memory.

In practice, the sink leg is often shorter than the source one. Also, source modules may produce output (e.g., error messages) and sink modules may request input (e.g., execution-time format commands.)

Another structure useful for implementing parts of a design is the transaction structure depicted in Figure 9. A "transaction" here is any event, record, or input, etc. for which various actions should result. For example, a command processor has this structure. The structure may occur alone or as one or more of the source (or even sink) modules of an input-process-output structure. Analysis of the transaction modules follows that of a transform module, which is explained later.

designing the structure

The following procedure can be used to arrive at the input-process-output general structure shown previously.

Step One. The first step is to sketch (or mentally consider) a functional picture of the problem. As an example, consider a simulation system. The rough structure of this problem is shown in Figure 10.

Step Two. Identify the external conceptual streams of data. An *external* stream of data is one that is external to the system. A *conceptual* stream of data is a stream of related data that is independent of any physical I/O device. For instance, we may have several conceptual streams coming from one I/O device or one stream coming from several I/O devices. In our simulation system, the external conceptual streams are the input parameters, and the formatted simulation the result.

Step Three. Identify the *major* external conceptual stream of data (both input and output) in the problem. Then, using the diagram of the problem structure, determine, for this stream, the points of "highest abstraction" as in Figure 11.

Figure 11 Determining points of highest abstraction

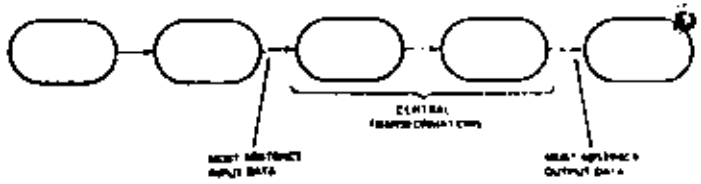
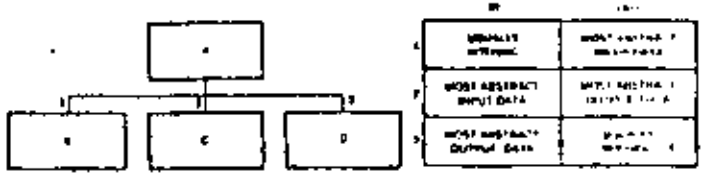


Figure 12 The top level



The "point of highest abstraction" for an input stream of data is the point in the problem structure where that data is farthest removed from its physical input form yet can still be viewed as coming in. Hence, in the simulation system, the most abstract form of the input transaction stream might be the built matrix. Similarly, identify the point where the data stream can first be viewed as going out—in the example, possibly the result matrix.

Admittedly, this is a subjective step. However, experience has shown that designers trained in the technique seldom differ by more than one or two blocks in their answers to the above.

Step Four. Design the structure in Figure 12 from the previous information with a source module for each conceptual input stream which exists at the point of most abstract input data; do sink modules similarly. Often only single source and sink branches are necessary. The parameters passed are dependent on the problem, but the general pattern is shown in Figure 12.

Describe the function of each module with a short, concise, and specific phrase. Describe what transformations occur when that module is called, not how the module is implemented. Evaluate the phrase relative to functional binding.

When module A is called, the program or system executes. Hence, the function of module A is equivalent to the problem being solved. If the problem is "write a FORTRAN compiler," then the function of module A is "compile FORTRAN program."

Module B's sufficient attribute obtaining the major stream of data. An example of a "typical module B" is "get next valid source statement in Polish form."

Module C's purpose is to transform the major input stream into the major output stream. Its function should be a nonprocedural description of this transformation. Examples are "convert Polish form statement to machine language statement" or "using keyword list, search abstract file for matching abstracts."

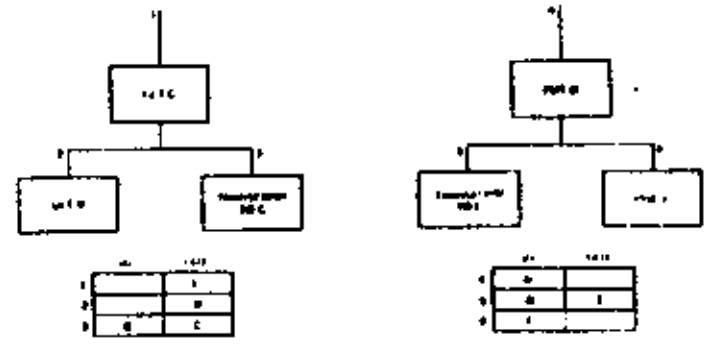
Module D's purpose is disposing of the major output stream. Examples are "produce report" or "display results of simulation."

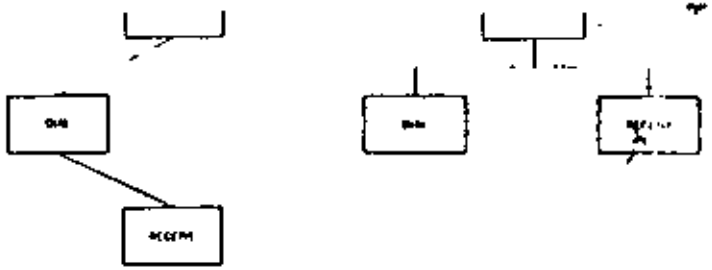
Step Five. For each source module, identify the last transformation necessary to produce the form being returned by that module. Then identify the form of the input just *prior* to the last transformation. For sink modules, identify the first process necessary to get closer to the desired output and the resulting output form. This results in the portions of the structure shown in Figure 13.

Repeat Step Five on the new source and sink modules until the original source and final sink modules are reached. The modules may be analyzed in any order, but each module should be done completely before doing any of its subordinates. There are, unfortunately, no detailed guidelines available for dividing the transform modules. Use binding and coupling considerations, size (about one page of source), and usefulness (are there subfunctions that could be useful elsewhere now or in the future) as guidelines on how far to divide.

During this phase, err on the side of dividing too finely. It is always easy to recombine later in the design, but duplicate func-

Figure 13 Lower levels





tions may not be identified if the dividing is too conservative at this point.

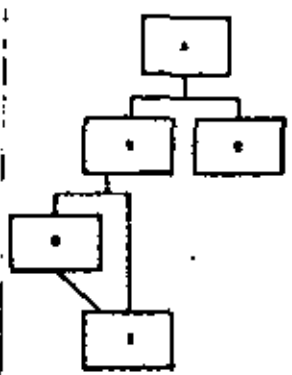
Design guidelines

The following concepts are useful for achieving simple designs and for improving the "first-pass" structures.

match program to problem

One of the most useful techniques for reducing the effect of changes on the program is to make the structure of the design match the structure of the problem, that is, form should follow function. For example, consider a module that dials a telephone and a module that receives data. If receiving immediately follows dialing, one might arrive at design A as shown in Figure 14. Consider, however, whether receiving is part of dialing. Since it is not (usually), have DIAL's caller invoke RECEIVE as in design B.

Figure 14 Scope of control



If, in this example, design A were used, consider the effect of a new requirement to transmit immediately after dialing. The DIAL module receives first and cannot be used, or a switch must be passed, or another DIAL module has to be added.

To the extent that the design structure does match the problem structure, changes to single parts of the problem result in changes to single modules.

scopes of effect and control

The *scope of control* of a module is that module plus all modules that are ultimately subordinate to that module. In the example of Figure 15, the scope of control of B is B, D, and E. The *scope of effect* of a decision is the set of all modules that contain some code whose execution is based upon the outcome of the decision. The system is simpler when the scope of effect of a decision is in the scope of control of the module containing the decision. The following example illustrates why.

flag to A or the decision will have to be repeated in A. The former approach results in added coding to implement the flag, and the latter results in some of B's function (decision X) in module A. Duplicates of decision X result in difficulties coordinating changes to both copies whenever decision X must be changed.

The scope of effect can be brought within the scope of control either by moving the decision element "up" in the structure, or by taking those modules that are in the scope of effect but not in the scope of control and moving them so that they fall within the scope of control.

module size

Size can be used as a signal to look for *potential* problems. Look carefully at modules with less than five or more than 100 executable source statements. Modules with a small number of statements may not perform an entire function, hence, may not have functional binding. Very small modules can be eliminated by placing their statements in the calling modules. Large modules may include more than one function. A second problem with large modules is understandability and readability. There is evidence to the fact that a group of about 30 statements is the upper limit of what can be mastered on the first reading of a module listing."

error and end-of-file

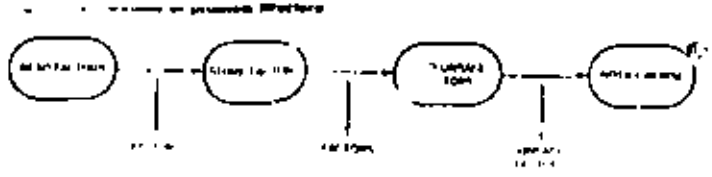
Often, part of a module's function is to notify its caller when it cannot perform its function. This is accomplished with a return error parameter (preferably binary only). A module that handles streams of data must be able to signal end-of-file (EOF), preferably also with a binary parameter. These parameters should not, however, tell the caller what to do about the error or EOF. Nevertheless, the system can be made simpler if modules can be designed without the need for error flags.

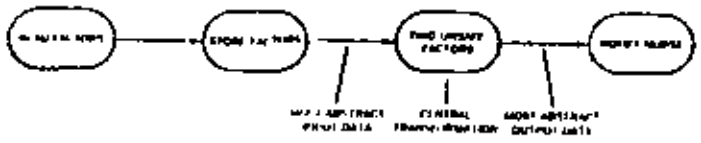
initialization

Similarly, many modules require some initialization to be done. An initialize module will suffer from low binding but sometimes is the simplest solution. It may, however, be possible to eliminate the need for initializing without compromising "black-boxness" (the same inputs *always* produce the same outputs). For example, a read module that detects a return error of file-not-opened from the access method and recovers by opening the file and rereading eliminates the need for initialization without maintaining an internal state.

selecting modules

Eliminate duplicate functions but not duplicate code. When a function changes, it is a great advantage to only have to change it in one place. But if a module's need for its own copy of a random collection of code changes slightly, it will not be necessary to change several other modules as well.





If a module seems almost, but not quite, useful from a second place in the system, try to identify and isolate the useful subfunction. The remainder of the module might be incorporated in its original caller.

Check modules that have many callers or that call many other modules. While not always a problem, it may indicate missing levels or modules.

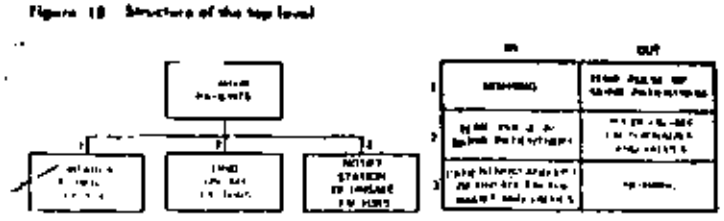
Isolate specifications

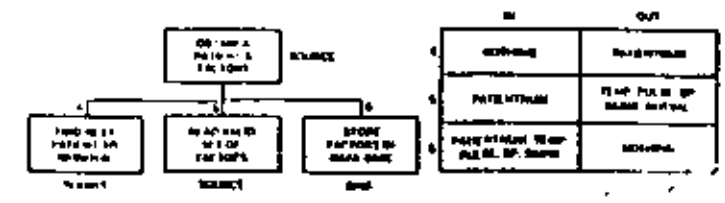
Isolate all dependencies on a particular data-type, record-layout, index-structure, etc. in one or a minimum of modules. This minimizes the recoding necessary should that particular specification change.

reduce parameters

Look for ways to reduce the number of parameters passed between modules. Count every item passed as a separate parameter for this objective (independent of how it will be implemented). Do not pass whole records from module to module, but pass only the field or fields necessary for each module to accomplish its function. Otherwise, all modules will have to change if one field expands, rather than only those which directly used that field. Passing only the data being processed by the program system with necessary error and EOF parameters is the ultimate objective. Check binary switches for indications of scope-of-effect/scope-of-control inversions.

Have the designers work together and with the complete structure chart. If branches of the chart are worked on separately, common modules may be missed and incompatibilities result from design decisions made while only considering one branch.





An example

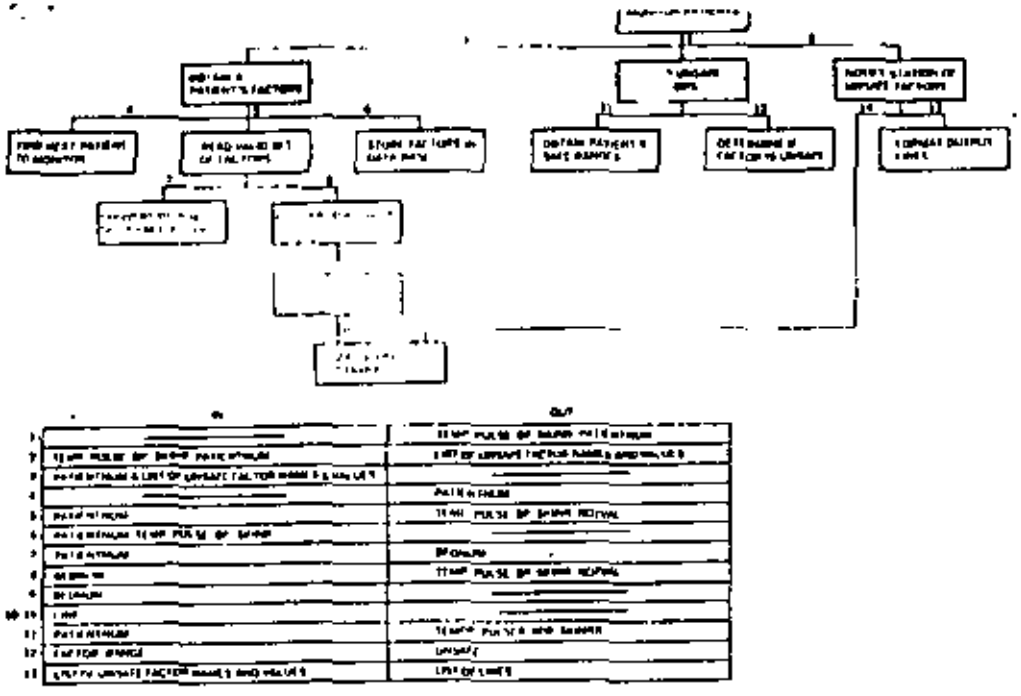
The following example illustrates the use of structured design:

A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient X's valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified.

In a real-life case, the problem statement would contain much more detail. However, this one is of sufficient detail to allow us to design the structure of the program.

The first step is to outline the structure of the problem as shown in Figure 16. In the second step, we identify the external conceptual streams of data. In this case, two streams are present, factors from the analog device and warnings to the nurse. These also represent the major input and output streams.

Figure 17 indicates the point of highest abstraction of the input stream, which is the point at which a patient's factors are in the form to store in the data base. The point of highest abstraction of the output stream is a list of unsafe factors (if any). We can now begin to design the program's structure as in Figure 18.



	IN	OUT
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		

In analyzing the module "OBTAIN A PATIENT'S FACTORS," we can deduce from the problem statement that this function has three parts. (1) Determine which patient to monitor next (based on their specified periodic intervals). (2) Read the analog device. (3) Record the factors in the data base. Hence, we arrive at the structure in Figure 19. (NOTVAL is set if a valid set of factors was not available.)

Further analysis of "READ VALID SET OF FACTORS", "FIND UNSAFE FACTORS" and "NOTIFY STATION OF UNSAFE FACTORS" yields the results shown in the complete structure chart in Figure 20

Note that the module "READ FACTORS FROM TERMINAL" contains a decision asking "did we successfully read from the terminal?" If the read was not successful, we have to notify the nurse's station and then find the next patient to process as depicted in Figure 21.

Modules in the scope of effect of this decision are marked with an X. Note that the scope of effect is *not* a subset of the scope

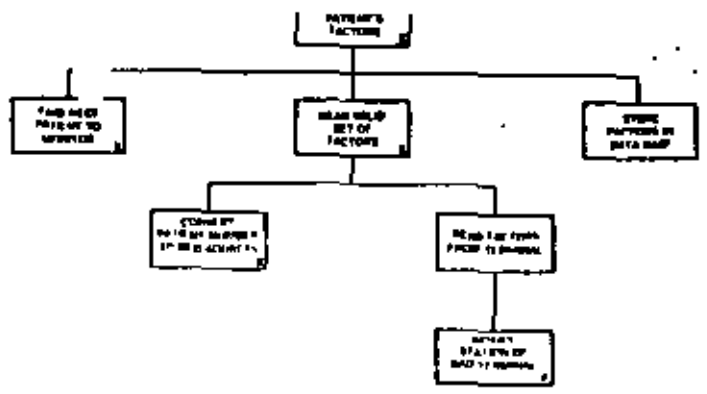
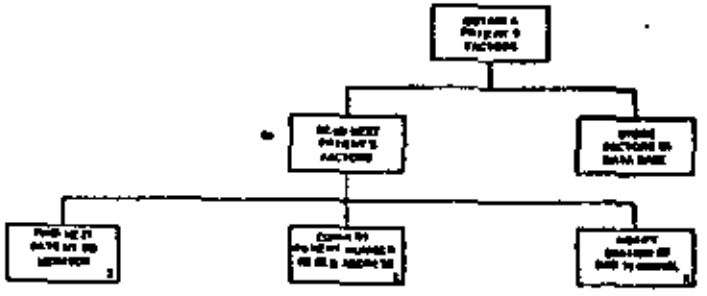


Figure 22 Scope of effect within scope of control



of control. To correct this problem, we have to take two steps. First, we will move the decision up to "READ VALID SET OF FACTORS." We do this by merging "READ FACTORS FROM TERMINAL" into its calling module. We now make "FIND NEXT PATIENT TO MONITOR" a subordinate of "READ VALID SET OF FACTORS." Hence, we have the structure in Figure 22. Thus, by slightly altering the structure and the function of a few modules, we have completely eliminated the problem.

Concluding remarks

The HIPO Hierarchy chart is being used as an aid during general systems design. The considerations and techniques presented here are useful for evaluating alternatives for those portions of the system that will be programmed on a computer. The charting technique used here depicts more details about the interfaces than the HIPO Hierarchy chart. This facilitates consideration during general program design of each individual connection and

should show them in the same block. However, the HIPO Hierarchy chart would still show all the functions in separate blocks.) The output of the general program design is the input for the detailed module design. The HIPO input-process-output chart is useful for describing and designing each module.

Structured design considerations could be used to review program designs in a walk-through environment.[11] These concepts are also useful for evaluating alternative ways to comply with the requirement of structured programming for one-page segments.[12]

Structured design reduces the effort needed to fix and modify programs. If all programs were written in a form where there was one module, for example, which retrieved a record from the master file given the key, then changing operating systems, file access techniques, file blocking, or DD devices would be greatly simplified. And if *all* programs in the installation retrieved from a given file with the same module, then one properly rewritten module would have *all* the installation's programs working with the new constraints for that file.

However, there are other advantages. Original errors are reduced when the problem at hand is simpler. Each module is self-contained and to some extent may be programmed independently of the others in location, programmer, time, and language. Modules can be tested before all programming is done by supplying simple "stub" modules that merely return preformatted results rather than calculating them. Modules critical to memory or execution overhead can be optimized separately and reintegrated with little or no impact. An entry or return trace-module becomes very feasible, yielding a very useful debugging tool.

Independent of all the advantages previously mentioned, structured design would *still* be valuable to solve the following problem alone. Programming can be considered as an art where each programmer usually starts with a blank canvas – techniques, yes, but still a blank canvas. Previous coding is often not used because previous modules usually contain, for example, *at least* GET and PUT. If the PUT is not the one needed, the GETs will have to be recoded also.

Programming can be brought closer to a science where current work is built on the results of earlier work. Once a module is written to get a record from the master file given a key, it can be used by all users of the file and need not be rewritten into each increasingly sophisticated systems.

Structured design concepts are not new. The whole assembly-line idea is one of isolating simple functions in a way that still produces a complete, complex result. Circuits are designed by connecting isolatable, functional stages together, not by designing one big interrelated circuit. Page numbering is being increasingly sectionalized (e.g., 4 – 101) to minimize the "connections" between written sections, so that expanding one section does not require renumbering other sections. Automobile manufacturers, who have the most to gain from shared system elements, finally abandoned even the coupling of the windshield wipers to the engine vacuum due to effects of the engine load on the performance of the wiping function. Most other industries know well the advantage of isolating functions.

It is becoming increasingly important to the data-processing industry to be able to produce more programming systems and produce them with fewer errors, at a faster rate, and in a way that modifications can be accomplished easily and quickly. Structured design considerations can help achieve this goal.

CITED REFERENCES AND FOOTNOTES

1. This method has not been submitted to any formal IBM test. Potential users should evaluate its usefulness in their own environment prior to implementation.
2. L. L. Constantine, *Fundamentals of Program Design*, in preparation for publication by Prentice-Hall, Englewood Cliffs, New Jersey.
3. G. J. Myers, *Composite Design: The Design of Modular Programs*, Technical Report TR00.2406, IBM, Poughkeepsie, New York (January 29, 1973).
4. G. J. Myers, "Characteristics of composite design," *Datamation* 19, No. 9, 100–102 (September 1973).
5. G. J. Myers, *Reliable Software through Composite Design*, to be published Fall of 1974 by Mason and Lipscomb Publishers, New York, New York.
6. HIPO – Hierarchical Input-Process-Output documentation technique. An adult education package. Form No. SR20-9413, available through any IBM Branch Office.
7. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* 11, No. 1, 56–73 (1972).
8. The use of the HIPO Hierarchy charting format is further illustrated in Figure 6, and its use in this paper was initiated by R. Ballow of the IBM Programming Productivity Techniques Department.
9. L. A. Belady and M. M. Lehman, *Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth*", RC 3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1971).
10. L. L. Constantine, "Control of sequence and parallelism in modular programs," *AFIPS Conference Proceedings, Spring Joint Computer Conference* 32, 409 (1968).
11. G. M. Weinberg, *PL/I Programming: A Manual of Style*, McGraw-Hill, New York, New York (1970).
12. *Improved Programming Technologies: Management Overview*, IBM Corporation, Data Processing Division, White Plains, New York (August 1973).

Directorio de Alumnos del curso: Metodología para el Desarrollo de Sistemas de Información 1982

1. José A. Aldana Benavides
Comisión Na . de .a Ind. Azucarera
Jefe de Serv. de Ingeniería
Morelos 104-2° Piso
Juárez
Cuauhtémoc
06140 México, D.F.
592 33 00 Ext.158

Amatista 8
Pedregal de Atizapán
Estado de Méx.
822 34 58

2. Clara A varado Zamorano
Centro de Instrumentos
UNAM
550 52 15 Ext. 4701

Paseo España 47
Lomas Verdes
III Secc. Naucalpan, Edo. de Méx.
562 98 34

3. Rubén Basurto Ríos
BANAMEX
Asesor
F. S.T. de Mier 143-10 Piso
Tránsito
México, D.F.
588 80 00 Ext.274

Puebla 353-203
Roma
6700 México, D.F.
286 55 08

4. Víctor H. Bueno Herrera
ICA, S.A.
Minería 145
Escandón
México, D.F.
516 04 60 Ext.618

Nte. 5A No.5006-1
Panamericana
07770 México, D.F.
567 60 38

5. Miguel A. Bravo García
UNAM

Sur 69 # 316 Depto.C 202
Ranjdal
Iztapalapa

6. Rigoberto Cabrera Ortega
IMP
Analista
Eje Central Lázaro Cárdenas 152
San Bartolo Atepehuacán
G. A. Madero
México, D.F.
567 66 00 Ext.2686

Cecilio Robelo 495
Jardín Ba buena
V. Carranza
15900 México, D.F.
784 55 80

7. Ricardo Cadena Galaicia
SARH
P. de la Reforma 107-7
San Rafael
Cuauhtémoc
527 06 05

Mar de Kara 18
Popotla
Miguel Hidalgo
México, D.F.

8. Carlos Campos Romero
SOFTWARE T.A., S.A.
Director de Sistemas
Reforma 403-603
Juárez
Cuauhtémoc
México, D.F.

Diego Abad 2
Circ. Pensadores
Cda. Satélite
572 25 27

9. Marina Castillo Garduño
SARH
Jefe de Secc.
Ofi. de Diagnóstico
Reforma 107-7 Piso
San Rafael
Cuauhtémoc
01000 México, D.F.
535 02 52

Doctor Durán 39-g
Doctores
Cuauhtémoc
06720 México, D.F.

10. Luis E. Castro Zarco
ISSSTE
Ponciano Arriaga 21 Altos
Tabacalera
México, D.F.
535 05 3

Avandoro 5
Vergel de Coyoacán
México, D.F.
677 31 4.

11. Lilia M. Chávez Romero
José Ma. Ollioqui 48
Del Valle
Benito Juárez
03100 México, D.F.
534 34 37 Ext.122

Calle D # 10 Manzana 2
Educación
Coyoacán
04400 México, D.F.
549 78 02

12. Ma. de L. Clares Fuentes
Fac. de Est. Sup. Cuautitlán
Carr. Cuautitlán Teoloyucán Km 10.5
54700 México
8720345 Ext.27

Peña Quebrada 31
Balcones del Valle
Tlanepant 1a, Edo. de Méx.
379 09 00

13. Ma. Cecilia Delgado Briseño de L.
UNAM
Centro de Instrumentos
Coyoacán
04510 México, D.F.
550 56 95

Ote. 162 No.232
Col. Moctezuma
V. Carranza
15500 México, D.F.
784 57 47

14. Carlos A. Ercick Espino
Facultad de Ingeniería
UNAM
México, D.F.
México, D.F.

Av.559 # 100
Unidad Aragón
G. A. Madero
551 28 32

15. Ricardo Escamilla Espinosa
Centro Nal. de Inf. y Estadísticas del Trabajo
Jefe del Depto. de Diseños de Sist. de Captación de Inf.
Bucareli 134-9° Piso
Centro
Cuauhtémoc
México,D.F.
512 19 79

16. Ricardo Espinosa Cerón
Centros de Integración Juvenil A.C.
José Ma. Oliaqui 48
Del Valle
B.Juárez
03100 México, D.F.
534 34 34

17. Francisco Ferrer Ferreira
ISSSTE
Analista
Ponciano Arriaga 31 Altos
Tabacalera
Cuauhtémoc
06030 México,D.F.
535 06 31

18. Eduardo Rafael Fernández Gutiérrez
Seguros América BANAMEX, S.A.
Av. Rev. 1508
Cpe. Inn
A.Obregón
01020 México,D.F.
550 99 99 Ext.2479

19. Jorge García Castañeda
Centros de Integración Juvenil
José Ma. Oliaqui 48
Del Valle
B.Juárez
03100 México, D.F.
534 34 35

20. Alejandro Luis E. Garduño Cañada
ISSSTE
Ponciano Arriaga 31
Tabacalera
Cuauhtémoc
México,D.F.
535 06 31

21. Cornelio Gómez Pérez
SAHOP
Av.Méx.-Xochimilco 5713
Tepepan
México,D.F.
676 46 46

Cal. de la Viga 735-205
Iztacalco
México,D.F.

Toltecas 56- Int. 5
San Javier Tlalnepantla,Edo. de México
13410 México, D.F.
565 08 88

Sierra Jiutepec 125
Lomas de Chapultpec
M.Hgo.
11010 México,D.F.
520 98 78

Rafael Delgado 101
Obrera
06800 México, D.F.

Cerro Gordo 150
Campestre Churubusco
Coyoacán
04200 México, D.F.
549 15 70

Sur 73 A No.226
Sinatel
Iztapalapa
09470 México, D.F.
581 74 24

Cristobal Colón Manzana 1 Lote 2
San Miguel Xalostoc
Edo. de Méx.

22. Sergio F. Glz. Pérez
I P N
Unidad Profesional Zacatenco
Lindavista
G.A.Madero
México, D.F.
586 47 11

Av. Fundidores 121
F. Del Hierro
Azcapotzalco
04600 México, D.F.
5671376

23. José L. Granados Reyes
Servi-Admón., S.A.
B.Franklin
Escandón
México, D.F.
277 10 44

U.Cuitláhuac 35 A 102
Nva. Sta. Ma.
México,D.F.
556 21 98

24. Alberto Guajardo Flores
Fac. de Ing.
UNAM
México,D.F.

Tortolas 8
Las Arboledas
Atizapán de Juárez
Edo. de México
379 27 80

25. Oscar Guillén Comenero
ISSSTE
Ponciano Arriaga 31 Altos
Tabacalera
Cuauhtémoc
06030 México, D.F.
535 06 31

Jacarandas 53
Fracc. Jacarandas
Edo. de Méx.
398 23 95

26. Fernando Guzmán Alvarez
Dir. Gral. de Est. del Territorio Nal.
S.A.Abad 124-1°
Tránsito
V.Carranza
México,D.F.
761 47 44

Yacatas 411 -7
Narvarte
B.Juárez
México, D.F.
536 75 64

27. Roberto Heatley Cortés

28. Miguel Islas Montaño
Distribuidora Conasupo Metropolitana
Km. 2 Carr. Pachuca-Tulancingo
42070 México
2 60 20

Manz.7 Lote 59
Progreso
42070 México

28. Alfonso López Reyes
DEPFI
UNAM
México,D.F.
550 52 15 Ext.4491

Av.Univ. 1900 Edif.20 Depto.202
B.Juárez
México,D.F.

29. Jesús R. Macías Muñiz
Bco. Mex. SOMEX
Gante 15-1
Centro
México,D.F.
518 18 00 Ext.161

Plan de Cpa. 56-9
Ticomán
México,D.F.
537 26 89

30. Jesús G. Mtz. Chavolla
Seguros América BANAMEX, S.A.
Av.Rev. 1508
Gpe. Inn
A.Obregón
01020 México,D.F.
550 99 99

Narciso Mendoza 40
A.Camacho
Naucalpán, Edo. de Méx.
53910 México, D.F.
589 56 10

31. Luis Mateos Sandoval
INFONAVIT
Bca. del Huerto 280
Gpe. Inn
01029 México, D.F.

Calle I-A no.22
S.José de la Escalera
D.G.A.Madero
07630

32. Luis R. Méndez Melgar
Cía. de Luz y Fza.
M. Ocampo 171
Anáhuac
México,D.F.
592 07 65

Casas Grandes 336
Narvarte
B.Juárez
México,D.F.
579 82 79

33. José G. Morales López
Grupo DICSA
Ibsen 15 Despacho 103
Chapultepec Morales
D.Cuauhtémoc
México,D.F.
540 03 30

1ra. Cerrada del Deportivo 2540
Texcoco, Méx.
4 29 10

34. Fernando Morales Lojero
Seguros América BANAMEX, S.A.
Analista
Av.Revolución 1508
Guadalupe Inn
A.Obregón
01020 México,D.F.
550 99 99

Av.Div. del Nte.3424
Xotepingo
Coyoacán
04610 México,D.F.
549 16 27

35. Guillermo Nava Vega
I M P
Eje Central Lázaro Cárdenas 152
San Bartolo Atepehuacán
G.A.Madero
07730 México,D.F.
567 66 00 Ext.2325

Calle 7 Torre 17,202
Unidad Alianza Popular Rev.
Coyoacán
04800 México,D.F.

36. José L. Ontiveros Poceros
Cía. de Luz y Fza. del Centro

37. Mario Palomar Alcibar
DEPFI
UNAM
Copilco
Coyoacán
04510 México,D.F.
550 52 11 Ext. 4493

Av.Cuauhtémoc 877-3
Narvarte
B.Juárez
00420 México,D.F.
543 78 85

38. Jorge Palafox Terán
Centro Nal. de Inf. y Estadísticas del Trabajo
Subdirector de Diseño
Bucareli 134-9°
Centro
06040 México,D.F.

Gral. Cano 58 Depto.103
San M. Chapultepec
D.M.Hgo.
11850 México, D.F.
271 38 42

39. José L. Patiño Donnadieu
Comisión Nal. Coordinadora de Ptos.
Cuernavaca No. 5
Condesa
México, D.F.
553 17 68

Ixtle 36
Las Palmas
México, D.F.
515 40 56

40. Guillermo A. Pérez Tabares
Dir. Gral. de Geografía del Territorio Nal.
S.A.Abad 124
Tránsito
México, D.F.
578 62 00 Ext. 169

Asistencia Pública 630-104
Federal
V.Carranza
15700 México, D.F.

41. Luis J. Poot Ayala
Dir. Gral. de Planificación
D D F
Pino Suárez 15-4°
Centro
Cuauhtémoc
México, D.F.
518 17 18

Dr. Vértiz 757-27
Narvarte
B.Juárez
03020 México, D.F.
530 29 95

42. Silvestre Revueltas Estrada
Celanese Mexicana, S.A.
Coordinador Proyectos D.Q.
Av. Rev. 1425
S. Angel Inn
A.Obregón
México,D.F.
548 69 60

Nda. Molino Flores 21
Bosque Echegaray
Edo. de México
560 99 85

43. Salvador Reyna Gómez
Golf 85
Country Club
Churubusco
Coyoacán
04210 México, D.F.
549 57 28

44. Clemente J. P.Rodríguez Barrón
DEPFI
UNAM
México, D.F.
550 52 15 Ext. 4493

José Rdz. Elz. 9
Constitución 1917
Iztapalapa
691 08 48

45. Arturo Rojas Flores
D D F
Pino Suarez No. 15
Centro
Cuauhtémoc
México, D.F.
518 17 18

Orquídeas 110
Villa de las Flores
Coacalco, Edo. de Méx.
874 08 29

6. Sergio Fco. Ruiz P.
DEPFI
UNAM
México, D.F.

Av. 9 # 37-3
Independencia
B.Juárez
9500 México, D.F.

47. Teodori hez García
ENEP Ac.. .n
Alcanfores y S.J. Totoltepec
Naucalpan, Edo. de Méx.
373 23 99

Mineros 62-5
Morelos
V.Carranza
México, D.F.

48. Jaime Taylor Torres
ISSSTE
Ponciano Arriaga 31 Altos
Tabacalera
Cuauhtémoc
México, D.F.
535 06 31

Moyobamba 333
Lindavista
G.A.Madero
México, D.F.
586 13 37

49. Claudio A. Tirado Osuna
Centro Nal. de Inf. y Estadísticas del Trabajo
Bucareli 134-9°
Centro
Cuauhtémoc
México, D.F.
585 66 00

Río Ganges 44-5
Cuau htémoc
México, D.F.
528 57 63

50. Felipe Toledo Mijangos
SAHOP
Xola y Universidad
Jefe de la Ofi. de Procesos de Datos
D.B.Juárez
México, D.F.
519 75 39

B. Fdz. A. 67
Los Cipreses
Coyoacán
México,D.F.
677 49 47

51. Odilón Torres Aguilera
FIRA Bco. de MEX.,S.A.
Insurgentes Sur 2375
S.Angel
A.Obregón
México, D.F.
550 70 11 Ext.245

Calle Ejido Culhuacán 43
S.Fco.Culhuacán
Coyoacán
México, D.F.

52. Luis R. Tovar Aguirr.
ISSSTE
Ponciano Arriaga 31
Tabacalera
Cuauhtémoc
México, D.F.
535 06 31

Río Consulado 1611
Peralvillo
Cuauhtémoc
México, D.F.
537 98 20

53. Humberto Vargas Díaz
U bdad de Planeación
Facultad de Ing.
UNAM
México, D.F.
550 52 15 Ext. 3707

Prosperidad A No. 148 Bis 'edcs.
Cam pestre Aragón
G. A. Madero
México, D.F.

54. Edgar S. Villazón Salem
Fac. de Ing.
Depto. de Ing. Industrial
México, D.F.

Casas Grandes 305
Narvarte
B.Juárez
03020 México, D.F.
590 29 89

55. Jose A. Zendejas Durán
ISSSTE
Ponciano Arriaga 31 Altos
Tabacalera
Cuauhtémoc
06030 México, D.F.
535 06 31

J. Loret. .ela 66
UnidadAragón
G.A.Madero
México, D.F.
551 02 91