

DIRECTORIO DE PROFESORES DEL CURSO: FACTORES ECONOMICOS DEL  
DESARROLLO DE SOFTWARE DICIEMBRE DE 1984 .

1. ING. DANIEL RIOS ZERTUCHE (COORDINADOR)  
DIRECTOR DE LA INFORMATICA  
SUBSECRETARIA DE PLANEACION  
DEL DESARROLLO  
S. P.P.  
IZAZAGA NO. 38-11° PISO  
MEXICO, D.F.  
521 98 98



FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE 1 9 8 4 .

Fecha	Tema	Horario	Profesor
Diciembre 3, 4, 5 6 y 7.	INTRODUCCION	17 a 21 h c/día	ING. DANIEL RIOS ZERTUCHE
	PERSPECTIVA ECONOMICA DEL CICLO DE VIDA DEL SOFTWARE		
	MODELOS SIMPLES DEL DESARROLLO DE SOFTWARE		
	FACTORES QUE INFLUENCIAN EL DESARROLLO DE SOFTWARE		
	SELECCION ENTRE ALTERNATIVAS		
	ANALISIS DE DECISIONES CON OBJETIVOS MULTIPLES		
	RIESGO, INCERTIDUMBRE Y VALOR DE LA INFORMA CION		
	TECNICAS PRACTICAS PARA LA ESTIMACION DE COSTOS.		
	CASOS PRACTICOS.		

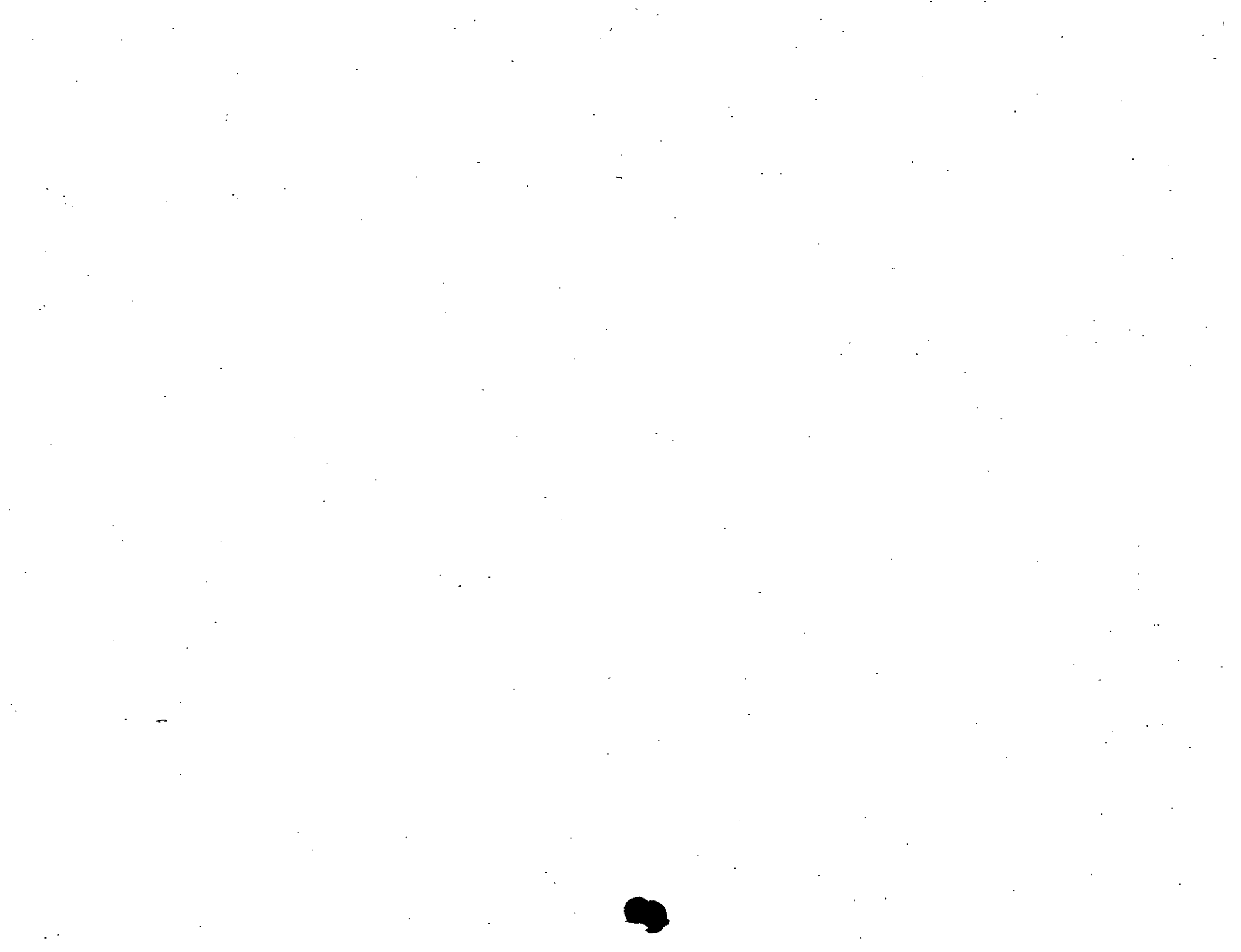


**DIVISION DE EDUCACION CONTINUA  
FACULTAD DE INGENIERIA U.N.A.M.**

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

SOFTWARE TECHNOLOGY IN THE 1990'S:  
USING AN EVOLUTIONARY PARADIGM

DICIEMBRE, 1984



---

*Improving software productivity is the key to reducing the rapidly widening gap between the demand for software and our ability to supply it.*

---

001

## Software Technology in the 1990's: Using an Evolutionary Paradigm

Barry W. Boehm, TRW

Thomas A. Standish, University of California, Irvine

In this article\* we offer a vision of the future.

It has become evident that there are serious problems to address concerning the production and support of computer software and that a significant effort is needed to improve the situation.

The demand for computer software is skyrocketing, and the means to meet this demand are glaringly inadequate. In the national economy, the need for software is growing rapidly as organizations strive to increase productivity through automation and to improve product versatility and market appeal through the use of computers.

In systems where software is a critical component, it is essential to obtain software that performs well, doesn't cost too much, and is maintainable over the projected system lifetime—in short, software that is *reliable, affordable, and adaptable*.

The central question we address in this article is how to achieve a state-of-practice in the 1990's where we can build embedded computer system software with adequate levels of *productivity, adaptability, and reliability*—that is, what will it take to achieve PAR by 1990?

To answer this question, we take a glimpse at the quantitative dimensions of the software demand-supply gap. How big is it? We cite several economic analyses of productivity that lead rather forcefully to the conclusion that there are no simple panaceas for the software problems we face. To the contrary, the economic analyses strongly indicate that we must make a wide-ranging, substantial attack on many constituent problems over a long period of time if we are to be successful. In short, the overall improvement we need must come from a large number of component improvements, each of which contributes meaningfully, but not overwhelmingly, to the whole.

We are convinced that success in combatting the software demand-supply gap can come only if we learn to manage a large number of variables skillfully and if the components to the overall solution integrate well. *Completeness and integration* are, therefore, two key concepts in our vision.

Many components of the software production infrastructure need to be improved and integrated, and we examine various stages of growth of the computer industry with an eye toward identifying catalytic forces that can be employed to stimulate the adoption of greater levels of shared infrastructure on which value-added production methods can be based and that allow for improved productivity.

This sets the stage for introducing the various elements of our vision of the future: how the Ada program can be shaped to serve as a catalyst for the introduction of highly integrated, complete software life-cycle support environments of high potential payoff; effective policies for technology insertion; policies for stimulating the adoption of greater levels of shared infrastructure; and how a proposed *Software Engineering Institute* can play a key role in collecting, integrating, and spreading improved tools, practices, and skills.

How might the future be different if the improved elements we have envisaged were in place and operating successfully in the 1990's? While we believe that no new *technological breakthroughs* will be needed to accomplish the improvements we seek, achieving *organizational breakthroughs* will be essential. Such organizational breakthroughs must succeed in collecting and integrating tools and in providing integrated support for software practices of proven effectiveness.

\*An earlier version of this article appeared as an appendix to the DoD Software Initiative strategy.

## Reducing the software supply-demand gap

A major technological concern confronting us in the 1990's is the serious and rapidly widening gap between the demand for software and our ability to supply it. As reported in Boehm<sup>2</sup> and Martin,<sup>3</sup> the national demand for software is rising by at least 12 percent per year, while the supply of people who produce software is increasing about four percent per year and the productivity of those software producers is increasing at about four percent per year; this leaves a cumulative four-percent gap.

Estimates of the current shortage exceed 100,000 software personnel. If this gap continues to widen at a rate of four percent per year, it will mean a shortfall of 800,000 to 1,000,000 software personnel by 1990.

A primary means of reducing this gap is to find ways to increase the productivity of the software people that we do have. Some extensive economic analyses of software productivity options, based on the Cocomo software cost-productivity model developed by Boehm,<sup>4</sup> have been done for TRW,<sup>5</sup> the US Army,<sup>6</sup> and the Department of Defense. Together, these studies reached the following positive conclusions:

(1) *Significant productivity gains require an integrated program of initiatives in several areas.* These areas include improvements in tools, methodology, work environments, education, management, personal incentives, and software reuse. A fully effective software support environment requires integration of software tools and office automation capabilities.

(2) *An integrated software productivity program can have an impressive payoff.* Productivity gains by factors of two in five years and factors of four in 10 years are generally achievable and are worth a good deal of planning and investment.

(3) *Improving software productivity involves a long, sustained effort.* While the payoffs are large, they require long-range commitment. There are no easy, instant panaceas.

Based on these conclusions, it appears that contemporary software organizations could initiate significant, long-range efforts to improve software productivity by

- a factor of two by 1988 and
- a factor of four by 1993

Even if we only achieved 70 percent of these gains, we could noticeably close the supply-demand gap by 1993.

The above analyses indicate that productivity increases of the required magnitude are achievable by sufficiently widespread pursuit of policies of judicious investment and by the adoption of software practices of proven effectiveness. Whether the leadership and collective will-power needed to accomplish this can be mustered is a different question. A national software initiative could play an influential role in stimulating awareness of the problem and its feasible solutions; such an initiative could also help organize the collaboration of enough people to close the gap. Although stiff challenges must be faced, success is clearly possible.

## Integrated software infrastructure for the 1990's

Successful software production rests on many layers of infrastructure. Figure 1 illustrates how applications software in DoD mission areas, such as avionics and command, control, communications, and intelligence, or C<sup>3</sup>I, relies on these layers of support infrastructure—for

002

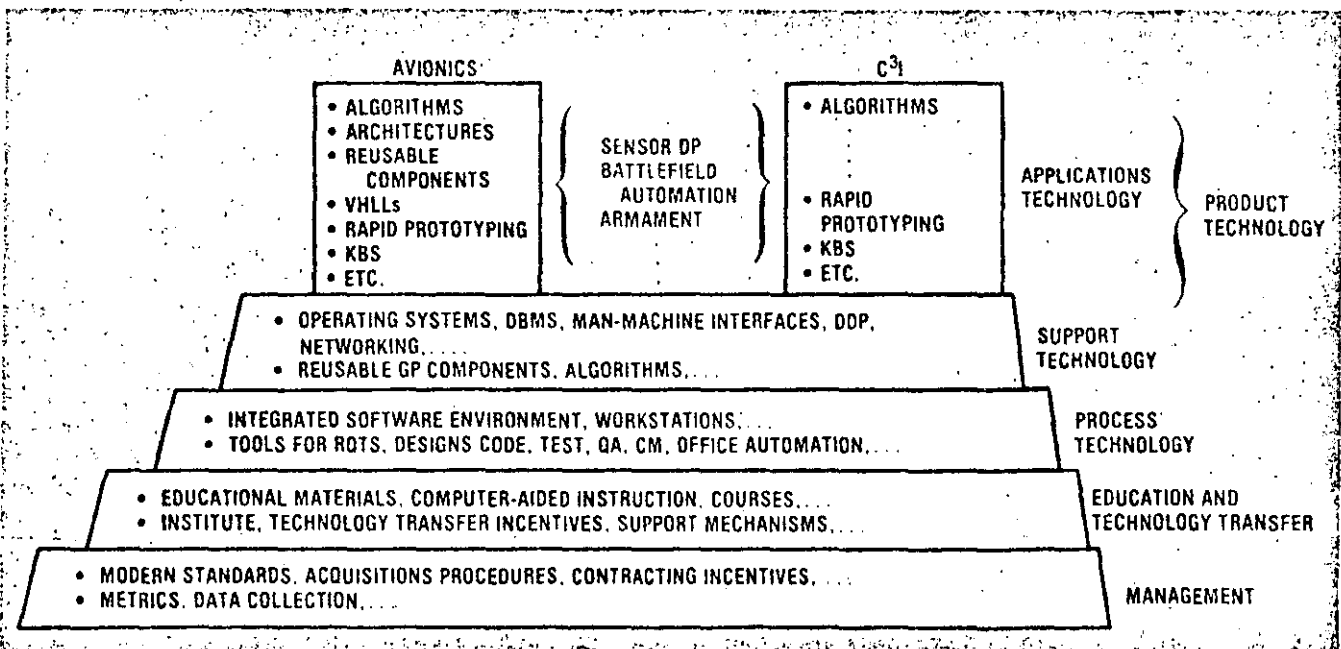


Figure 1. Integrated software infrastructure for the 1990's.

example, on application-specific technology, general-purpose support technology, process technology, education, technology transfer capabilities, and management skills.

Suppose we envisage a future software support environment encompassing all of the elements in Figure 1; suppose, also, that these elements are well-integrated. How would such a support environment differ from the environments we have today?

In point of fact, nearly all of the components of the environment illustrated in the figure have some sort of implementation today. However, there are two main problems with these existing components. First, they are largely immature and incompletely developed. Examples here are rapid prototyping capabilities, requirements aids, APSE (Ada programming support environment) master databases, portable KAPSEs (kernel APSEs), etc. Second, the components are poorly integrated—for example, tools supporting different phases of the life-cycle; obsolete acquisition regulations, specifications, and standards; environments requiring the mastery of 12 distinct tool command languages to do one's job; people not trained or qualified to use advanced capabilities, etc.

If we could correct these shortfalls in the existing DoD and support contractor environment, most of the DoD's current software problems could be eliminated. Further-

more, as the discussion in the previous section has suggested, the DoD could reap an estimated software life cycle productivity gain of a factor of four if it could achieve the fully integrated environment depicted in Figure 1. Such analyses and conclusions apply equally well, we believe, to nonmilitary computer technology.

Thus, a key element in our vision of the state of software practices in the 1990's is the attainment of a fully populated, fully integrated software environment that models the structure illustrated in Figure 1.

Certainly, some of the software practices effort in the current decade must be devoted to developing new techniques and capabilities in areas such as rapid prototyping, requirements aids, reusable software component libraries, etc. But the major portion of it must be a carefully planned and coordinated program of enhancing and refining existing technical, personnel, managerial, and institutional capabilities to fit within, and perform as, a unified environmental structure. In short, we need to define and integrate complete support environments and put them into widespread practice.

Before we describe some further elements of our vision of the 1990's having to do with the catalytic role we envisage for the Ada program and the role a properly conceived Software Engineering Institute could play, we need to make some observations about forces in the computing marketplace. It is essential to understand the nature and evolution of these forces in order to master the technology transfer processes that we believe are required for success.

Table 1.  
Growth stages in the computer industry.

STAGE/DECADE	SHARED INFRASTRUCTURE	VALUE-ADDED ISSUES
I 1950	None	Shared Utilities Media Standards
II 1960	Shared Utilities Media Standards Algorithms	MOL/HOL Mixed Peripherals I/O Standards
III 1970	HOLs Vendor Utilities I/O Standards Plus Previous Base	Software Unbundling Stable OS Interface
IV 1980	Vendors-Line OS, Utilities, Plug-Compatible Main- frames, Commercial Soft- ware Packages, Basic Soft- ware Environments, Plus Previous Base	Portable OS Environment Networking Standards
V 1990	Ada Portable OS, Utilities Portable Environments Networking Standards Some Mainframe Standards Nebula, 1750	Application Standards Mainframe Standards
VI 2000	Knowledge-Based Applica- tion Standards, Program Generators, Component Libraries for Some Areas	Application Standards for More Complex Knowledge Domains

### The computer industry environment: stages of growth

The dependence of significant productivity gains on a high level of shared support infrastructure has been demonstrated for a number of industrial advances in the past; see, for example, the article by Graham.<sup>7</sup> The computer industry has gone through several stages of growth—stages in which higher levels of shared infrastructure led to increases in productivity. With the attainment of each new level, a successful resolution of the conflict between the following two opposing forces in the computing marketplace took place:

- (1) An antisharing, antistandardization force motivated by the desire to retain one's existing customer base, and
- (2) A prosharing, prostandardization force motivated by the desire and need to improve user's productivity in order to keep them as customers.

To progress from one infrastructure level to the next requires two primary ingredients. First, a technology that is sufficiently mature to support standardization and improved productivity and, second, a set of catalysts to get the improvement process far enough along to overcome the antisharing forces.

Table 1 presents six stages of growth in the computer industry, each roughly corresponding to a decade. These are characterized in terms of the shared infrastructure available to computer users at each stage and the primary



value-added standardization versus installation-uniqueness issues being resolved by the industry at the time.

Thus, for example, around 1950, there was no real shared infrastructure, but the technology was beginning to raise issues such as

- Should punch card formats be standardized in the interests of data sharing, or should multiple (80-column, 90-column, etc.) formats continue and proliferate further?
- Given that Lockheed, Douglas, and North American are competitors, should they share mathematical routines and system utilities to promote common productivity?

By 1960, these issues were largely resolved in favor of shared infrastructure and improved productivity. By this time, though, similar new issues had arisen over such items as the use of HOLs and I/O standards that allowed users to choose between several options for peripheral devices.

Right now, the computer industry is roughly at Stage IV in its evolution. Its shared infrastructure currently includes some vendor-line operating systems and utilities, allowing the choice of plug-compatible mainframes and commercial software packages, and the beginnings of portable operating systems and environments such as Unix and CP/M.

Some of the major sharing versus installation-unique issues prevalent today are

(1) Should mainframe vendors support portable operating systems and software environments, thus providing users access to more productivity options at the risk of losing users tied to vendor-specific operating systems, utilities, and software environments?

(2) In particular, should those be based on Ada?

(3) Should software houses developing their own software environments (for competitive advantages or software product sales potential) merge their products with common-use standard environments?

(4) Should the industry establish at least interim standards for local area network interfaces?

There are many strong, conservative, antisharing forces around today that could easily cause the balance on these issues to tip toward installation-specific proliferation and reduced value-added productivity potential in the US. This is a matter of some national concern, for it is likely that our worthiest international competitors will opt for shared use and higher productivity, thus giving themselves a real competitive edge in the computer industry.

The proposed DoD STARS program is this country's major opportunity to tip the balance on the software-related issues discussed above in the direction of shared use and higher productivity. Via maturation of Ada-based operating systems, utilities, and software environments and a set of catalysts (R&D contracts, technology transfer activities, and system contract incentives), STARS can ensure that the technology base evolves into the necessary shared infrastructure.

In our vision of the future, we see a state of practice in the 1990's that embraces the use of complete software life-cycle environments that improve productivity and, at the same time, improve our capacity to produce software that is reliable and adaptable. To get from here to there, we will need to adopt increased levels of shared infrastructure, we will have to integrate much of what is possible to do in isolation today, and we will have to achieve effective technology insertion of the resulting environment.

004

### Our vision of the 1990's

For the purposes of this article, think of the STARS program as the umbrella under which all of what follows happens; also consider the Ada Program and the proposed Software Engineering Institute (SEI) to be parts of it.

**Software productivity policies.** While software producers have ample incentive for attempting to do something on their own about software productivity (for, among other reasons, attempting to maintain a competitive posture in the industry), some national leadership in this matter would greatly help. In fact, the maintenance of the current US technological lead in software may depend critically on national leadership in matters of productivity, and failure to address this issue at the national level may mean forfeiting that lead to other nations.

We envisage that the STARS program will take some action to achieve the desired property of software "affordability" in the 1990's. As we have shown, we feel that vigorous pursuit of productivity improvement policies could achieve this aim.

Guidelines for setting up an in-house productivity project could be issued by the DoD. These would be in the same spirit as the current DoD CAD/CAM initiative to develop guidelines and support capabilities to improve the productivity of DoD equipment manufacturers.

For software, an initial set of such guidelines has already appeared in the literature.<sup>2</sup> As part of the STARS program, productivity project guidelines called, say, Productionman could be drafted. These guidelines could then be circulated to the computer software community at large and could be fine tuned in the fashion of Steelman, Stoneman, Methodman, and Educationman. Appendices could provide a few case histories, and the SEI could serve as the laboratory for evolving a working model free for all to adopt. While several corporations, such as IBM (Santa Teresa) and TRW, currently have such projects, an explicit model in the public domain—one that has the right characteristics for effective technology insertion—could provide important advantages for software development in the US in the 1990's.

We foresee that the DoD could formulate and adopt an effective means to spread software productivity policies nationally by (1) producing guidelines in a form like Productionman under a STARS program activity, (2) making the Software Engineering Institute a living showcase for Productionman and ensuring that an easy, effective technology insertion path for it is provided, and (3) seeing

005  
to it that DoD procurement policies require the adoption of effective software productivity plans by contractors.

In our vision of the software world of the 1990's, we predict that effective leadership from the STARS program will help bring about our scenario wherein 70 percent of the software producers will have quadrupled their productivity by 1993. By making this productivity level an important goal and by providing the guidance and incentives to accomplish it, the STARS program could make it happen. Without focus, leadership, and the will to do it, however, it may not happen.

**Software procurement policies.** We foresee that the STARS program will study *procurement policies* and will draft, tune, and cause to be adopted some procurement policies that are to be used in connection with government-procured Ada software. Some elements of such policies are already under discussion with respect to the requirement to use certified Ada compilers on Ada programming projects.

As new techniques mature, procurement policies may have to be revised to take advantage of associated benefits. For example, if rapid prototyping techniques emerge to enable contractors to buy information that reduces risk in the early stages of a software project, some revision of procurement policies might be advisable to allow prototyping to be incorporated into the development cycle (without the contractor's being accused of "coding before writing his B5 specs."). The current, initial DoD use of competitive concept definition contracts is a good step in this direction, but it is not yet well supported by DoD policies and procedures. Here is another example: If techniques are developed that allow the reuse of reliable software components, some incentives may need to be built into the procurement process to encourage contractors to reuse components rather than rebuilding their own employing (otherwise profitable) cost-plus practices.

**The role of the Ada program.** The important thing about Ada, in our opinion, is not so much the *product* as the *process*. The process involves getting a wide community of sometimes competing interest groups to come together, formulate an agreement about *sharing* a way of doing business, and then living with it and reaping its benefits. The product, Ada, is a medium of exchange in which future sharing and cooperation can take place. Ada may represent a critical new element of shared infrastructure facilitating value-added commerce.

In our view of the 1990's software world, we envisage that the Ada process could be used repeatedly in the 1980's to achieve shared agreements and provide a shared infrastructure to enhance productivity. Some possible examples are (1) portable, standard programming environments—mature APSEs with complete life-cycle support toolsets,<sup>8</sup> (2) an Ada software reuse library and retrieval system, (3) a standard "a-code," or Ada machine code, which could be cheaply reproduced on new target machines (in, say, a man-month of labor), and which, like Pascal's p-code, could provide a machine-independent way for supporting Ada software but which could also be microcoded or implemented in

silicon for greater performance, and (4) APSE virtual workstation standards and local network protocol standards to enable simple porting of Ada application software without having to change interface and network communication protocols.

We also foresee that the SEI could spearhead this effort by pulling together pieces of the Ada program, such as the ones just described, actively soliciting community input and managing the evolution of related proposed standards and shared agreements, providing a showcase of how it all works, and providing the means to integrate the technology into receiving organizations.

**The Software Engineering Institute.** We envisage the founding of a new, national-level institute to carry out improvements critical to attaining the state of practice needed in the 1990's.

We believe that existing institutions can't do the job. A powerful catalyst with considerable influence is needed—something that can provide a bridge between existing organizations in government, industry, and academia; something capable of bringing these existing institutions into cooperative endeavors. We doubt this can be accomplished from within.

---

### **The Software Engineering Institute could act as an integrative agent to achieve a consensus on critical new aspects of shared infrastructure.**

---

An institute like the SEI could play several influential roles. It could collect and integrate existing tools into common, unified, software life-cycle support frameworks. Such an institute could act as an integrative agent by energetically soliciting community opinion and helping the community to achieve a consensus on critical new aspects of shared infrastructure for the 1990's. Additionally, it could furnish effective institutional support for the technology insertion process. This process needs to be carefully planned and managed, and if it is not vigorously supported, an essential part of the overall job cannot be accomplished.

A key feature of technology transfer is to have people from the DoD, industry, and academia rotating through the institute. Such rotation would have the additional benefit of keeping the institute fresh and vital over time. It would thus be a magnet for top talent without being a talent sink.

Technology insertion happens gradually and may be the single biggest obstacle in the way of the STARS program's success. Organizations will not adopt new technology unless (1) they hear about it and have the opportunity to see it work; (2) they know it is within their economic means to acquire it; (3) it is demonstrated to their satisfaction (either by in-house pilot projects or credible demonstrations elsewhere) that it yields good cost-benefit improvements; and (4) it can be readily learned. While some technology insertion happens without any explicit leadership or management, we envisage

that the STARS program will stand a better chance of success if its technology insertion components are explicitly planned, managed, and funded. Here, the SEI can play a big role, and our vision of the 1990's holds that this is exactly what it will do.

The SEI should select one or more key application areas and apply the institute's advanced Ada methods to the actual construction of Ada software systems for real target computers. In the process, the SEI should demonstrate vastly improved productivity (compared to current baselines), should write the software for *adaptability* and *reliability* (and spearhead advances in our knowledge of how to produce Ada software with these characteristics), should derive from its created application a library of Ada software components (to provide a basis for software reuse), and should support technology insertion. The SEI can do all of this by

- publicizing its work and methods and giving demonstrations,
- making sure its methods and equipment are affordable for others to acquire, thus paving the way for acquisition,
- giving statistics on its projects that demonstrate great cost-benefit results,
- providing course materials to educate people in the use of Ada methodologies and APSE toolsets, and
- through the use of rotating assignments of personnel on a two-way street from the institute to the organization receiving the technology, engage in actual technology insertion activities.

Thus, the most important mission and function of the institute, after shaking down Ada methods, is its technology transfer of those methods into appropriate receiving organizations.

It will be essential to have an affordable Ada-based method for rehosting APSEs. Our vision of the future isn't going to work if we lack an effective means to rehost Ada software and if vigorous institutional support for technology insertion cannot be counted on.

### Putting it all together

Suppose the various parts of the vision we have sketched become a reality and are in place and operational in the 1990's. How will the world be different? Let's list some things we believe will differ dramatically from the world today.

First, software manufacturing in this country would be vastly more *productive* than it is now. In fact, it would be over 300 percent more productive by 1993. This could be the case, not at the sacrifice of software quality, but rather with the simultaneous achievement of improved *adaptability* and *reliability*—in short, we would have achieved PAR, and the software demand gap would be closed.

Second, we would have in place (or at least in the initial stages of technology insertion, to be honest) a set of shared agreements that would form the backbone and infrastructure for a flourishing commerce and exchange of

computer software. The Ada language would be a widespread medium of exchange and libraries of Ada software components would set the stage for an improved economic system of *value-added production* in which software components could be acquired in the marketplace, assembled into new products, and then sold in the marketplace after value had been added.

Third, the Software Engineering Institute would be a showcase and a technology insertion agent for effective, integrated, complete software environments supporting key application areas. It would illustrate the new methods and techniques, quantify their benefits, construct credible cases showing that the new methods and technologies in fact work well, assist in technology insertion into receiving institutions, and spearhead educational efforts to spread the new way of doing business.

Fourth, improved procurement policies would provide guidance and incentives for software reuse. They would also allow for the incorporation of prototyping into contracts to buy information to reduce risk, and would encourage the spread of productivity improvements; these improvements would, perhaps, be seen initially in the defense contractor community, but would later spread to the entire commercial software community as successful implementations proliferate.

Fifth, educational materials would be available to train practitioners in the use of good tools and effective practices.

Sixth, and finally, all of these capabilities would be integrated into the cohesive framework of the "smoke-stack" diagram given in Figure 1. The easy, consistent, mutually reinforcing use of these capabilities across the entire software life cycle will improve the productivity, adaptability, and reliability of the software maintenance process and products at least as significantly as it will impact software development.

This world of the 1990's does not require an unknown technological breakthrough to achieve its significant PAR gains.\* What it does require is a lot of thoughtful, carefully coordinated hard work to improve all the components of our existing software process and environment in ways that make them more individually effective and mutually reinforcing. ■

### Acknowledgment

The contribution of Thomas Standish to the preparation of this document was supported by DARPA under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project (ARPA work order AO4365). The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

\*On the other hand, we believe investments to search for technological breakthroughs in software technology are certainly worthwhile and could produce even more significant PAR gains.

## Appendix

### Improving software productivity: an economic analysis

007

How can we address the problem of increasing software productivity? One approach is to analyze the cost drivers that contribute to the cost of software products. Some of these cost drivers are controllable, and by investing to provide software project resources or by following selected policies and disciplines, we can control factors that improve productivity.

For example, the constructive cost model, or Cocomo, described in detail by Boehm,<sup>4</sup> estimates the cost of a software project as a function of the program size in delivered source instructions (DSI) and a number of other cost-driver attributes summarized in the figure below.

This figure shows the productivity range for each attribute; in other words the relative productivity in DSI/man-month ascribable to the given attribute after normalizing for the effects of other attributes. Thus, the 1.49 productivity range for the *software tools* attribute results from an analysis indicating that, all other factors being equal, a project with a very high level of tool support will require only 0.83 of the effort required for a project with a nominal level of tool support, while an equivalent project with a very low level of tool support would require 1.24 times the effort required for the nominal project, or  $1.24/0.83 = 1.49$  times the effort on the "very high" tools project. The "very high" and "very low" ratings and multipliers correspond to specific levels on a Cocomo rating scale for tool support<sup>4</sup> and to the results of analyzing a database of 63 software projects.

Given rating scales for the software tools attribute and for the other cost driver attributes, it is possible to perform a *productivity audit* of a software project to determine the weighted-average productivity multipliers characteristic of the project in relation to the nominal mea-

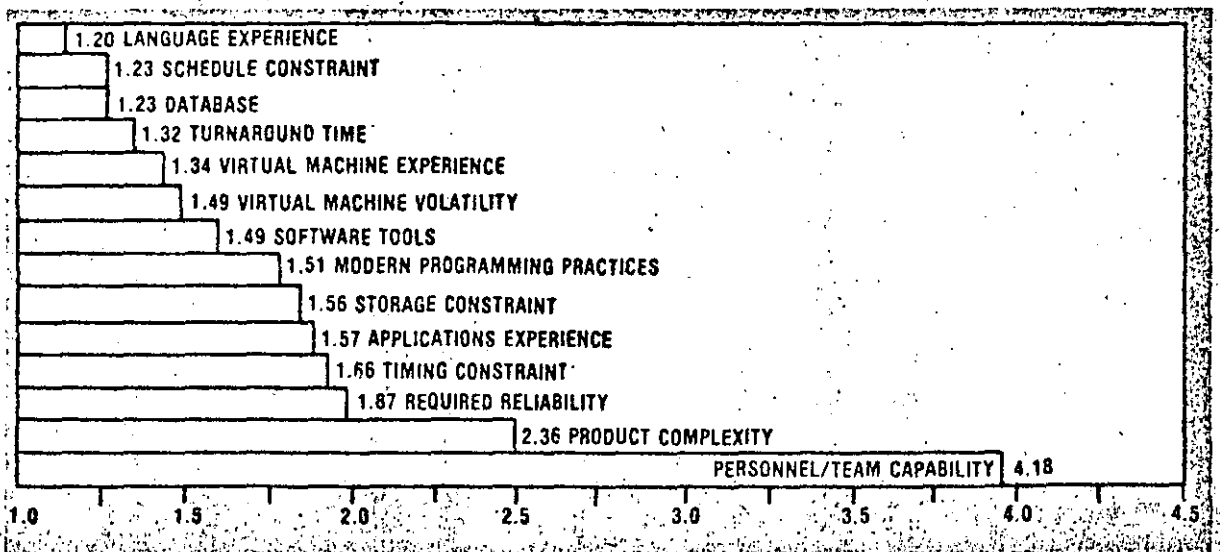
asures in the Cocomo model. Then, one can investigate several future scenarios representing varying levels of investment into productivity-improving measures. The table below summarizes the analysis of the potential impact of the software initiative on DoD software productivity.<sup>1</sup> The 1976 multipliers are the average of the DoD projects in the 63-project Cocomo database; the 1983 multipliers are an estimate of current practice; and the 1993 multipliers are an estimate of the results possible with a major DoD software initiative.

The table shows that a productivity improvement program that simultaneously realizes several cost-driver attribute improvements could improve productivity by a factor of 4.34 between 1983 and 1993. (The potential gain for maintenance may be overstated somewhat, as the last three factors apply less to maintenance than to development. On the other hand, the lower Cocomo maintenance multipliers for modern programming practices partly compensate for this effect.)

Besides providing an estimated productivity gain, the analysis gives insights for determining which tools and policies to emphasize in a selected productivity improve-

Estimated DoD software initiative impact  
on software productivity.

Cost Driver	DoD Average-Effort Multipliers		
	1976	1983	1993
Use of Software Tools	1.05	1.02	0.85
Use of Modern Programming Practices	0.98	0.95	0.85
Programming Language Experience	1.03	1.02	0.98
Software Environment Experience	1.05	1.03	0.95
Computer Execution Time Constraint	1.25	1.18	1.11
Computer Storage Constraint	1.22	1.15	1.06
Computer Turnaround Time	1.03	1.01	0.90
Reduced Requirements Volatility	1.17	1.15	1.00
Retool Avoidance	1.06	1.06	1.00
Software Reuse	0.93	0.90	0.50
Relative Effort	2.01	1.54	0.36
Productivity Gain		1.30	4.34



ment strategy. Moreover, it provides a valuable framework for tracking the actual progress of a productivity program and for determining whether its goals are actually being achieved.

## References

1. L. E. Druffel, *Strategy for a DoD Software Initiative*, CSS DUSD(RAT), Washington, DC, 1982.
2. B. W. Boehm, "Improving Software Productivity," *Proc. Compton Fall 81*, Washington, DC, 1981.
3. E. W. Martin, "Strategy for a DoD Software Initiative," *Computer*, Vol. 16, No. 3, Mar. 1983, pp. 52-59.
4. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
5. B. W. Boehm et al., "The TRW Software Productivity System," *Proc. Sixth Int'l Conf. Software Engineering*, Tokyo, Japan, pp. 148-156.
6. M. W. Alford et al., "Distributed Computer Design Study," TRW report to USA-BMDATC, 1981.
7. A. K. Graham, "Software Design: Breaking the Bottleneck," *IEEE Spectrum*, 1982.
8. T. A. Standish, "The Importance of Ada Programming Support Environments," *AFIPS Conf. Proc.*, Vol. 51, 1982 NCC, pp. 333-339.



**Barry W. Boehm** is chief engineer of TRW's Software and Information Systems Division. He is currently managing three TRW programs focusing on software productivity, research, and cost methodology. He is also a visiting professor at UCLA. Previously, he headed The Rand Corporation's Information Sciences Department.

For the IEEE Computer Society, he served formerly as chairman of the Technical Committee on Software Engineering and currently serves on the IEEE Computer Society Governing Board. He is also a member of the AFIPS Social Implications Committee. His most recent book is *Software Engineering Economics*.

Boehm received his BA in mathematics from Harvard in 1957 and his MA and PhD degrees from the University of California at Los Angeles in 1961 and 1964, respectively.



**Thomas A. Standish** is a professor of computer science at the University of California, Irvine. He is also chairman of the board of the Irvine Computer Sciences Corporation. Currently, he serves on the IEEE Technical Committee on Software Engineering. Previously, Standish was chairman of the Irvine Computer Science Department. Before that, he taught at Harvard and Carnegie-Mellon and was a senior scientist at Bolt Beranek and Newman, Inc. He also served as editor-in-chief of the *ACM Monograph Series* and as programming languages editor for *Communications of the ACM*. In 1980, he published the book *Data Structure Techniques*.

Standish received a PhD in computer science from Carnegie Institute of Technology in 1967 and a BS in mathematics (magna cum laude) from Yale in 1962.

Questions about this article can be directed to Thomas A. Standish, Computer Science Department, University of California, Irvine, Irvine, CA 92717.

# 018 AT WESTERN ELECTRIC, WE TEACH UNIX\* OPERATING SYSTEMS AS IF WE INVENTED THEM.

## Announcing training from the creators of the UNIX Operating Systems.

\* Now everyone can get top quality UNIX Operating Systems training, from the people who created them—Western Electric and Bell Laboratories. Along with our certified instructors, we offer a complete curriculum for UNIX Operating Systems, including UNIX System V.

These courses are the same as those conducted internally at Western Electric and Bell Labs. And we furnish them at your location or at one of our conveniently located centers: Princeton, NJ; Chicago, IL; Columbus, OH; and Sunnyvale, CA.

We provide an individual terminal for each student. And in the evening, the use of our facilities and terminals is available at no extra cost. In addition, volume discounts are available.

All UNIX Operating Systems courses are designed and developed to high quality standards by Western Electric and Bell Labs, as part of a total commitment to UNIX Operating Systems Support. Now with our training, you can learn firsthand, what everyone else has been teaching secondhand.

For information, call us at 800-221-1647 or write to Western Electric, P.O. Box 2000, Hopewell, NJ 08525.



**AT&T**  
Western Electric



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

STRUCTURED PROGRAMMING IN A PRODUCTION  
PROGRAMMING ENVIRONMENT

DICIEMBRE, 1984

# Structured Programming<sup>1</sup> in a Production Programming Environment

F. TERRY BAKER

**Abstract**—This paper discusses how structured programming methodology has been introduced into a large production programming organization using an integrated but flexible approach. It next analyzes the advantages and disadvantages of each component of the methodology and presents some quantitative results on its use. It concludes with recommendations based on this generally successful experience, which could be useful to other organizations interested in improving reliability and productivity.

**Index Terms**—Chief programmer teams (CPT's), development support libraries (DSL's), structured coding, structured programming, top-down development, top-down programming.

## I. INTRODUCTION

AT this point in time, the ideas of structured programming have gained widespread acceptance, not only in academic circles, but also in organizations doing production programming. What is perhaps not so widely appreciated, however, is that the organizations, procedures, and tools associated with the implementation of structured programming are critical to its success. This is particularly true in production programming environments, where program systems (rather than single programs) are developed, and the attainment of reliable, maintainable software on time and within cost estimates is a prime management objective. In this environment, module level coding and debugging activities typically account for about 20 percent of the effort spent on software development [1]. Thus, narrow applications of structured programming ideas limited only to these activities have correspondingly limited benefits. It is therefore desirable to adopt an integrated but flexible approach incorporating the ideas into as many aspects of projects as possible to achieve maximum reliability improvements and cost savings.

## II. BACKGROUND

The IBM Federal Systems Division (FSD) is an organization involved in production programming on a large scale. Although much of its software work is performed for federal, state, and local governmental agencies, the division also contracts with private business enterprises for complex systems development work. Work scope ranges from less than a man-year of effort on small projects to thousands of man-years spent on the development and maintenance of large, evolutionary, long-term systems such as the Apollo/Skylab ground support

software. Varying customer requirements necessitate the use of a wide variety of hardware, programming languages, software tools, documentation procedures, management techniques, etc. Problems range from software maintenance, through pure applications programming using commercially available operating systems and program products, to the concurrent development of central processors, peripherals, firmware, support software and applications software for avionics requirements. Thus, within this single organization can be found a wide range of software development efforts.

FSD has always been concerned with the development of more reliable programs through use of improved software tools, techniques and management methods. Most recently, FSD has been active in the development of structured programming techniques [2]. This has led to organizations, procedures and tools for applying them to production programming projects, particularly with a new organization called a chief programmer team (CPT) [3]. The team, a functional organization based on standard support tools and disciplined application of structured programming principles, had its first trial on a major software development effort in 1969-71 [4], [5]. In the three years since the completion of that experimental project, FSD has been incorporating structured programming techniques into most of its software development projects. Because of the scope and diversity of these projects, it was impossible to adapt any single set of tools and procedures or any rigid type of organization to all or even to a majority of them. And because of the ongoing nature of many of these systems, it was necessary to introduce these techniques gradually over a period of several years. It is believed that any software development organization can improve the reliability and reduce the costs of its software projects using an approach similar to that described herein.

## III. PLAN

To introduce the ideas of structured programming into FSD work practices and to evaluate their use, a plan with four major components was implemented. First, a set of guidelines was established to define the terminology associated with the ideas with sufficient precision to permit the introduction and measurement of individual components of the overall methodology. These guidelines were published, and directives regarding their implementation were issued. Second, support tools and methodologies were developed, particularly for projects using commercial hardware and operating systems. For those projects

Manuscript received February 1, 1975.

The author is with IBM Federal Systems Division, Gaithersburg, Md. 20760.

Reprinted from *IEEE Transactions on Software Engineering*, June 1975. Copyright © 1975 by The Institute of Electrical

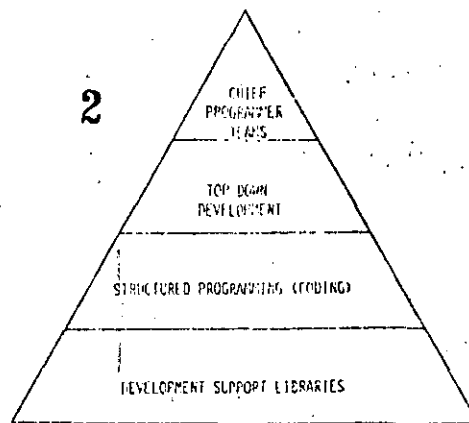


Fig. 1. Hierarchy of techniques.

where these were not employed, standards based on the developed tools enabled them to provide their own support. Third, documentation of the techniques and tools, and education in their use, were both carried out. These were done on a broad scale covering management techniques, programming methodologies, and clerical procedures. Fourth, a measurement program was established to provide data for technology evaluation and improvement. This program included both broad measurements which were introduced immediately, and detailed measurements which required substantial development work and were introduced later. The next four subsections cover the components of this plan and their implementation in detail.

#### A. Guidelines

A number of important considerations influenced the establishment of a set of guidelines for the application of structured programming technology within FSD. First and most important, they had to permit adaptation to the wide variety of project environments described above. This required that they be useful in program maintenance situations where unstructured program systems were already in being, as well as in situations where completely new systems were to be developed. Second, they had to allow for the range of processors and operating systems in use. This necessitated the description of functions to be provided instead of specific tools to be used. Third, they had to allow for differences in organizations and methodology (e.g., specifications, documentation, configuration management) required or in use on various projects.

The guidelines resulting from these considerations describe a hierarchical set of four components, graphically illustrated in Fig. 1. With one limited exception, use of the component at any level presupposes use of those below it, even though certain components could be used more independently. Thus, by beginning at a level which a project's environment and status permit, and then progressing upward, projects can evolve gradually toward full use of the technology.

1) *Development Support Libraries (DSL's)*: The intro-

ductory level is the DSL which is a tool designed with two key principles in mind.

1) Keep current project status organized and visible at all times. In this way, any programmer, manager or user can find out the status or study an approach directly without depending on anyone else's interpretation.

2) Make it possible for a librarian to do the majority of library maintenance, thus separating clerical from intellectual activity.

A DSL is normally the primary responsibility of a secretary or clerk trained as a programming librarian. Programmers interface with the computer primarily through the library and the programming librarian. This allows better control of computer activity and ensures that the library is always complete and current, thus reducing misunderstandings and inconsistencies. In general, the library system is the prime factor in increasing the visibility of a developing project and thus reducing risk and increasing reliability.

The guidelines for a DSL are as follows.

1) A library system providing the capabilities described in Section III-B 1) below must be used.

2) The library system must be used throughout the development process, not just to store debugged source or object code, for example.

3) Visibility of the current status of the entire project, as well as past history of source code activities and run executions, should be provided by the external library.

4) The visibility of the code should be such that the code itself serves as the prime reference for questions of data formats, program operation, etc.

5) Filing procedures must be faithfully adhered to for all runs, whether or not setup is performed by a librarian. However, use of a trained librarian is recommended.

2) *Structured Coding*: In order to provide for use of structured programming techniques on maintenance as well as development projects, it was necessary to separate them into two components. In FSD, then, we distinguish between those practices used in system development (top-down development) and those used in



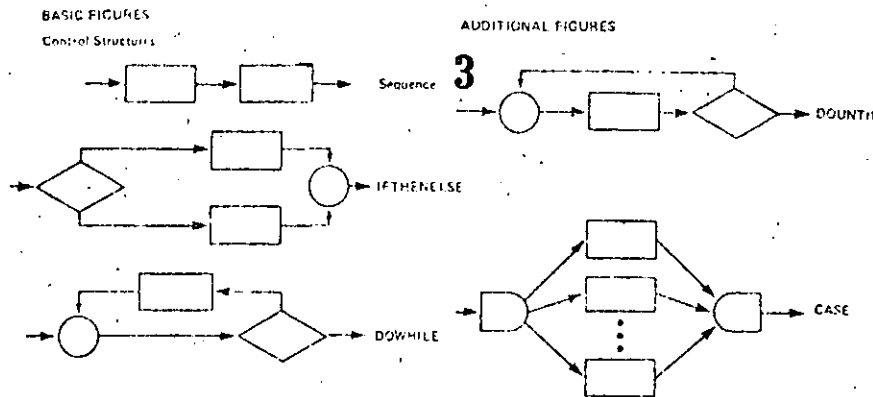


Fig. 2. Control structures.

coding individual program modules (structured coding). Our use of the term "structured coding" in the guidelines thus refers primarily to standards governing module organization and construction, and control flow within it. The three basic control flow figures and two optional ones shown in Fig. 2 are permitted. The guidelines also refer to a Guide [6] (see Section III-C below) which contains general information and standards for structured coding, as well as detailed standards for use of various programming languages. Finally, they require that code be reviewed by someone other than the developer. The detailed guidelines for structured programming are as follows.

1) The conventions established in the *Structured Programming Guide* should be followed. Exceptions to conventions must be documented. If a language is not covered in the Guide, then use of a locally generated set of conventions consistent with the rules of structured programming is acceptable.

2) The code should be reviewed for functional integrity and for adherence to the structured programming conventions.

3) A DSL must be used.

3) *Top-Down Development:* Top-down development refers to the process of concurrent design and development of program systems containing more than a single compilable unit. It requires development to proceed in a way which minimizes interface problems normally encountered during the integration process typical of "bottom-up development" by integrating and testing modules as soon as they are developed. Other opportunities provided are for the following.

1) A project to staff up more gradually and reduce the total manpower required.

2) Computer time requirements to be spread more evenly over the development period.

3) The user to work major portions of the system much earlier and identify gross errors before acceptance testing.

4) Most of the system to be used long enough by the time it is delivered that both the user and the developer have confidence in its reliability.

The term "top-down" may be somewhat misleading taken too literally. What top-down development really implies in everyday production programming is that one

builds the system in a way which ideally eliminates (or more practically, minimizes) writing any code whose testing is dependent on other code not yet written, or on data which are not yet available. This requires careful planning of the development sequence for a large system consisting of many programs and data sets, since some programs will have to be partially completed before other programs can be begun. In practice, it also recognizes that exigencies of customer requirements or schedule may force deviations from what would otherwise be an ideal development sequence. The guidelines for top-down development are as follows.

1) Code currently being developed should depend only on code already operational, except in those portions where deviations from this procedure are justified by special circumstances.

2) The project schedule should reflect a continuing integration, as part of the development process, leading directly to system test; as opposed to a development, followed by integration, followed by system test cycle.

3) The managers of the effort should have attended a structured programming orientation course (see Section III-C below).

4) Structured coding and a development support library system must be used. (Because ongoing projects may not be able to install a DSL, an implementation of only structured coding is acceptable in these cases.)

4) *CPT's:* A CPT is a functional programming organization built around a nucleus of three experienced persons doing well-defined parts of the programming development process using the techniques and tools described above. It is an organization uniquely oriented toward the techniques and tools and is a logical outgrowth of their introduction and use. Described in detail in [3]-[5], it has been used extensively in FSD on projects ranging up to approximately 100,000 lines of source code and is being experimented with on larger projects. The guidelines for CPT's are as follows.

1) One person, the chief programmer, should have complete technical responsibility for the effort. He should ordinarily be the manager of the other people.

2) There must be a backup programmer prepared to assume the role of chief programmer.

3) Top-level code segments and the critical control paths of lower level segments should be coded by the chief and backup programmers.

4) Other programmers should be added to the team only to code specific well-defined functions within a framework established by the chief and backup programmers.

5) The chief and backup programmers must review the code produced by other members of the team.

6) Top-down development, structured coding, and a DSL must be used.

### B. Support

Support of several types is necessary in order to permit effective implementation of, and achieve maximum benefits from, the ideas of structured programming. Development support libraries, introduced above, are a recognized and required component of the methodology employed in FSD. Standards are necessary to ensure a consistent approach and to help realize benefits of improved project communications and manageability. Procedures are required for effective use of the tools and to permit functional breakup and improved overall efficiency in the programming process. Finally, other techniques of design, programming, testing, and management can be helpful in a structured programming environment as well as in a conventional one.

1) *DSL's*: The need for and value of DSL's both as a necessity for structured programming and as a vehicle for project communication and control, has been thoroughly covered in [3]-[7]. Early work on DSL's in FSD centered on the provision of libraries for projects using IBM's System/360 Operating System and Disk Operating System in batch programming development situations. Subsequent work on DSL's in FSD has extended the support to some of the non-System/360 equipment in use and also introduced interactive DSL's for use both by librarians and programmers [8]. Furthermore, a study of general requirements for DSL's has been performed under contract to the United States Air Force and has been published in [9]. DSL's are now available for and in use on most programming projects in FSD.

A DSL keeps *all* machine readable data on a project—source code, object code, linkage editor language, job control language, test data, and so on—in a series of data sets which comprise the internal library. Since all data are kept internally and are fully backed up, there is no need for programmers to generate or maintain their own personal copies. Corresponding to each type of data in the internal library, there is a set of current status binders which comprise the external library. These are filed centrally and used by all as a standard means of communication. There is also a set of archives of superseded status pages which are retained to assist in disaster recovery, and a set of run books containing run results. Together, these record the activities—current and historical—of an entire project and keep it completely organized.

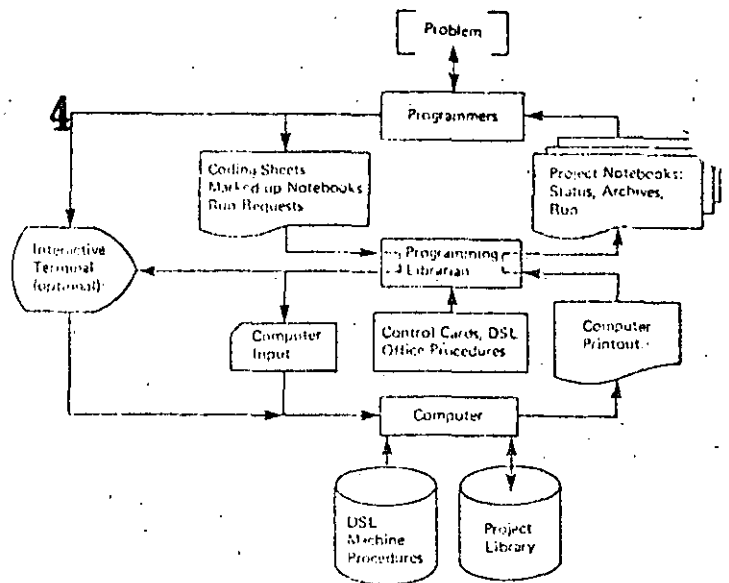


Fig. 3. DSL operations.

The machine procedures, as the name implies, are cataloged procedures which perform internal library initiation, updating, compilation and test, housekeeping and termination. Most of them are used by programming librarians by means of simple control cards they have been trained to prepare.

The office procedures are a set of "clerical algorithms" used by the programming librarian to invoke the machine procedures, to prepare the input and file the output. Once a new code has been created and placed in the library initially, a programmer makes additions and corrections to it primarily by marking up pages in the external library and giving them to the programming librarian to make up control and data cards to cause the corresponding changes or additions to be made to the internal library. As a result, clerical effort and wasted time on the part of the programmers are significantly reduced. Fig. 3 shows the work flow and the central role of the programming librarian in the process. Machine and office procedures for a typical DSL are documented for programmers in [6] and for librarians in [10].

2) *Standards*: To support structured coding in the various languages used, standards were required. These covered the implementation of the control flow figures in each language as well as the conventions for formatting and indenting programs in that language.

There are four approaches which can be taken to provide the basic and optional control flow figures in a programming language, and each was used in certain situations in FSD.

1) The figures may be directly available as statements in the language. In the case of PL/I, all of the basic figures were of this variety. In Cobol, the `IFTHENELSE` (with slight restrictions) and the `DOUNTIL` (as a `PERFORM`) were present.

2) The figures may be easily simulated using a few standard statements. The `CASE` statement may be readily

simulated in PL/I using an indexed GOTO and a LABEL array, with each case implemented via a DO-group ending GOTO to a common null statement following all cases.

3) A standard preprocessor may be used to augment the basic language statements to provide necessary features. The macro assembler has been used in FSD to add structuring features to System/360, System/370 and System/7 Assembler Languages.

4) A special precompiler may be written to compile augmented language statements into standard ones, which may then be processed by the normal compiler. This was done for the Fortran language, which directly contains almost none of the needed features.

The result of using these four approaches was a complete set of figures for PL/I, Cobol, Fortran, and Assembler. Using these as a base, similar work was also done for several special-purpose languages used in FSD.

To assist in making programs readable and in standardizing communications and librarian procedures, it was desirable that programs in a given language should be organized, formatted and indented in the same way. (This was true of the Job Control and Link Editor Languages as well as of the procedural languages mentioned above.) Coding conventions were developed for each covering the permitted control structures, segment formatting, naming, use of comments, labels, and indentation and formatting for all control flow and special (e.g., OPEN, CLOSE, DECLARE) statements.

5) *Procedures:* An essential aspect of the use of DSL's is the standardization of the procedures associated with them. The machine procedures used in setting up, maintaining and terminating the libraries were mentioned above in that connection. However, the office procedures used by librarians in preparing runs, executing them and filing the results are also quite extensive. These were developed and documented [10] in a form readily usable by nonprogramming oriented librarians.

4) *Other:* While the above constitute the bulk of the work originated by FSD, certain other techniques and procedures have been assimilated into the methodology in varying degrees. These include management techniques, hierarchy plus input-process-output (HIPO) diagrams and structured walk-throughs.

FSD has been active in the development of management techniques for programming projects. A book [1] resulting from a management course and guide used in FSD has become generally available. As top-down development and structured coding came into use, it became apparent that traditional management practices would have to be substantially revised (see Section IV-C below). An initial examination was done, and a report [12] was issued which has been very valuable in guiding managers

o using the new methodology. Some of this material has now been added to a revised edition of the *FSC Programming Project Management Guide* [13] from which the book mentioned above was drawn.

A technique called HIPO diagrams [7], [14] developed by the IBM System Development Division (SDD) has

5 proved valuable in design and documentation in top-down development. HIPO consists of a set of operational diagrams which graphically describe the functions of a program system from the general to the detail level. Not to be confused with flowcharts, which describe procedural flow, HIPO diagrams provide a convenient means of documenting the functions identified in the design phase of a top-down development effort.

Structured walk-throughs [7] were developed on an SDD CPT project as a formal means for design and code reviews during the development process. Using HIPO diagrams and eventually the code itself, the developer "walks through" his efforts for the reviewers. These latter may consist of the chief or backup programmer (or lead programmer if a CPT is not being employed), other programmers and a representative from the group which will formally test the programs. Emphasis is on error avoidance and detection, not correction, and the attitude is open and nondefensive on the part of all participants (today's reviewer will be tomorrow's reviewee). The reviewers prepare for the walk-through by studying the diagrams or code before the meeting, and follow-up is the responsibility of the reviewee, who must notify the reviewers of corrective actions taken.

### C. Documentation and Education

Once the fundamental tools and guidelines were established, it was necessary to begin disseminating them throughout FSD. Much experimental work had already been done in developing the tools and guidelines themselves, so that a cadre of people familiar with them was already in being.

Most of the documentation has been referred to above. The primary reference for programmers was the *FSC Structured Programming Guide* [6]. In addition to the standards for each language and for use of DSL's, it contained general information on the use of top-down development and structured coding, as well as the procedures for making exceptions to the standards when necessary. It also contained provisions for sections to be added locally when special-purpose languages or libraries were in use. Distributed throughout FSD, the Guide has been updated and is still the standard reference for programmers. The *FSC Programming Librarian's Guide* serves a similar purpose for librarians and also has provisions for local sections where necessary. While the use of the macros for System/360 Assembler Language was included in the *Programming Guide*, additional documentation [15] was available on them if desired. Finally, management documentation in the form of [11]-[14] was also available.

It was recognized that providing documentation alone was not sufficient to permit most personnel to begin applying the techniques. A series of courses (one for each major language) was set up to train experienced FSD programmers in structured programming and DSL techniques. Lasting 25 hours, these courses provided

instruction and, more importantly, practice problems which forced the programmers to begin the transition process. Once all programmers had been introduced to the ideas these courses were discontinued, and structured programming is now included as part of the basic programmer training courses given to newly hired personnel.

The same situation held true for managers as well as programmers. Because FSD wished to apply the methodology as rapidly as possible, it was desirable to acquaint managers with it and its potential immediately. Thus, one of the first actions taken was to give a half-day orientation course to all FSD managers. This permitted them to evaluate the depth to which they could begin to use it on current projects, and to begin to plan for its use on proposed projects. This was then followed up by a 12-hour course for experienced programming managers, acquainting them with management and control techniques peculiar to top-down development and structured coding. (It was expected that most of these managers would also attend one of the structured programming courses described above to acquire the fundamentals.) Again, now that most programming managers have received this form of update, the material has now been included in the normal programming management course given to all new programming managers.

#### *D. Measurement*

One of the problems of the production programming world is that it has not developed good measures of its activities. Various past efforts, most notably the System Development Corporation studies [16] have attempted to develop measurement and prediction techniques for production programming projects. The general results have been that a number of variables must be accurately estimated to yield even rough cost and schedule predictions, and that the biggest factors are the experience and abilities of the programmers involved. Nevertheless, it was felt in FSD that some measures of activity were needed, not so much for prediction as for evaluation of the degree to which the methodology was being applied, the reliability and productivity improvements which were achieved and the problems which were experienced in its use. To these ends, two types of measurements were put into effect.

The first type of measurement, implemented immediately, was a monthly report required from each programming project. In addition to some quantitative data on his project, each manager was required to state the following:

- 1) the total number of programmers on the project;
- 2) the number currently programming;
- 3) the number using structured coding;
- 4) the number of programming groups on the project;
- 5) the number of CPT's;
- 6) whether a DSL was in use; and
- 7) whether top-down development was in use.

These figures were summarized monthly for various levels of FSD management and were a valuable tool in

ensuring that the methodology was indeed being introduced, as well as that the guidelines were being followed.

The second type of measurement was a much more comprehensive one. It required a great deal of research in its preparation, and eventually took the form of a questionnaire from which data were extracted to build a measurement data base. The questionnaire contains 105 questions organized into the following eight sections:

- 1) identification of the project;
- 2) description of the contractual environment;
- 3) description of the personnel environment;
- 4) description of the personnel themselves;
- 5) description of the technical environment;
- 6) definition of the size, type and reliability of the programs produced;
- 7) itemization of the financial, computer and manpower resources used in their development; and
- 8) definition of the schedule.

The questionnaire is administered at four points during the lifetime of every project. The first point is at the beginning, in which all questions are answered with estimates. The next administration is at the end of the design phase, when the initial estimates are updated as necessary. It is again filled out halfway through development, when actual figures begin to be known. And it is completed for the last time after the system has been tested and delivered, and all results are in. The four points provide for meaningful comparisons of estimates to actuals, and allow subsequent projects to draw useful guidance for their own planning. The data base permits reports to be prepared automatically and statistical comparisons to be made.

## IV. IMPLEMENTATION EXPERIENCE

Each of the four components of the methodology which FSD has introduced has resulted in substantial benefits. However, experience has also revealed that their application is neither trivial nor trouble free. This section presents a qualitative analysis of the experience to date, describing both the advantages and the problems.

### *A. Development Support Libraries*

Most projects of any size have historically gravitated toward use of a program library system of some type. This was certainly true in FSD, which had some highly developed systems already in place when the methodology was introduced. These were primarily used as mechanisms to control the code, so that differing versions of ongoing systems could be segregated. In some cases they provided program development services such as compilation, testing, and so forth. However, none were being used primarily to achieve the goals of improved communications or work functionalization which are the direct benefits of a DSL. In fact, the general attitude toward the services they provided was that they were there to be

used when and if the programmers wished. Most code in them was presumed private, with the usual exceptions of macro and subroutine libraries.

One of the most difficult problems in the introduction of the DSL approach was to convince ongoing projects that their present library systems fulfilled neither the requirements nor the intents of DSL. A DSL is as much a management tool as a programmer convenience. A programming librarian's primary responsibility is to management, in the sense of supporting control of the project's assets of code and data—analogue to a controller's responsibility to management of supporting control of financial assets. The project as a whole should be entirely dependent on the DSL for its operation, and this, more than any other criterion, is the determining factor in whether a library system meets the guidelines as a DSL.

When all functions are provided, and a project implements a DSL, then a high degree of visibility is available. Programmers use the source code as a basic means of communication and rely on it to answer questions on interfaces or suggest approaches to their problems. Managers use the code itself (together with the summary features of more sophisticated DSL's) to determine the progress of the work. Users also benefit, even at an early stage of implementation, from the ready availability of the test data and the possibility of using part of the developing system on an experimental basis without interference with or from the rest.

The visibility in itself is valuable, even on a laissez-faire basis. But when it is coupled with well-managed code-reading procedures, it also provides reliability improvements. The walk-throughs described above, or equivalent procedures, ensure that someone in addition to the developer reviews the code, verifying that the specifications have been addressed, checking the planned test coverage, assisting in standards compliance and, last but not least, constructively criticizing the content. While the review procedure is obviously greatly facilitated by concomitant use of structured programming, it is possible without it and was included with the DSL guidelines to encourage its adoption.

The archives which are an integral part of a DSL provide an ability to refer to earlier versions of a routine—sometimes useful in tracing intent when a program is passed from hand to hand. More importantly, they give a project the ability to recover from a disaster in which part of its resources are destroyed. (It is perhaps obvious but worth mentioning that this will not be complete insurance unless project management sees to it that the backup data sets are stored physically separate from the working versions.) There was an initial tendency in FSD to over-collect and to over-formalize the archiving process. It appears unnecessary to retain more than a few generations of object code, run results and so forth. The source code and test data generally warrant longer retention, but even here it rapidly becomes impractical to save all versions. In general, sufficient archives should be retained to provide complete recovery capability when

used in conjunction with the backup data sets, plus enough additional to provide back references.

The separation of function introduced by the DSL office procedures has two main benefits. The obvious one is of lowered cost through the use of clerical personnel instead of programmers for program maintenance, run setup and filing activities. A significant additional benefit comes about through the resulting more concentrated use of programmers. By reducing interruptions, librarians afford the programmers a work environment in which errors are less likely to occur. Furthermore, they permit programmers to work on more routines in parallel than typically is the case.

The last major benefit experienced from a DSL rests in its support of a programming measurement activity. By automatically collecting statistics of the types described above, they can enhance our ability to manage and improve the programming process. The early DSL's in FSD did not include measurement features, and the next generation is only beginning to come into use, so a full assessment of this support is not yet possible.

It was difficult to convince FSD projects in some cases that a well-qualified programming librarian could benefit a project as much as another programmer. In fact, there was an initial tendency to use junior programmers or programmer technicians to provide librarian support. This had two disadvantages and hence is not recommended. First, the use of programming-qualified personnel is not necessary because of the well-defined procedures inherent in the DSL's. Use of over-qualified individuals in some cases led to boredom and sloppy work with a resulting loss of reliability. Second, such personnel cannot perform other necessary functions when needed. One of the advantages of using secretaries as librarians is that they can use both skills effectively over the lifetime of a typical project. During design and documentation phases, they can provide typing and transcription services; while during coding and testing phases, they can perform the needed librarian work.

Related to this is the need to provide backup librarian services. In most cases this has been accomplished through informal cross-project sharing or through temporary assumption of the duties by programmers.

Two problems remain in defining completely the role of librarians. First, the increasing use of interactive systems for program development is forcing an evolution of librarian skills toward terminal operation and test support rather than coding of changes and extensive filing. The most effective division of labor between programmer and librarian in such an environment remains to be determined. It also appears possible to use librarians to assist in documentation, such as in preparation of HIPO diagrams. Second, FSD has a number of small projects in locations remote from the major office complexes and support facilities—frequently on customer premises. Here it is not always possible to use a librarian cost-effectively. In this situation, better definition of the programmer-librarian relationship in the interactive system develop-

ment environment may permit some degree of development and librarian support from the central facility instead of requiring all personnel to be on-site.

### B. Structured Coding

8

Structured coding was separated from top-down development primarily to permit ongoing projects to use some of the methodology. Combined with usage of a DSL, it provides enhanced readability of code, enforces modularity (and thus encourages changeability) and maintainability, simplifies testing, and permits improved manageability and accountability. These are all well-known benefits and need not be elaborated on here. An additional, unplanned for, result of structured coding is that it tends to encourage the property of "locality of reference," which improves performance in a virtual systems environment.

The introduction of structured coding was not easily achieved in FSD. The broad variety of projects, languages and support has already been mentioned, and the development of DSL's, the Guides [6], [10], and the education program were necessary before widespread application of the methodology could take place. Furthermore, the ongoing nature of many of the systems meant that structured coding could take place only as modules were rewritten or replaced.

This gradual introduction created a problem of education timing. Practically, it was most expedient to have programmers attend the education courses between assignments. The nature of the courses was such that they introduced the techniques and provided some initial practice. Yet they required substantial work experience using the techniques to be fully effective. Structured coding requires the development of a whole new set of personal patterns in programming. Until old habits are unlearned and replaced by new ones, it is difficult for programmers to fully appreciate its advantages. For best results, this work experience and the overcoming of the natural reluctance to change habits should follow the training immediately. This was not always feasible and resulted in some loss of educational effectiveness.

A second problem arose because of the real-time nature of a significant fraction of FSD's programming business. Here the difficulty was one of demonstrating that structured coding was not detrimental to either execution speed or core utilization. While it is difficult to verify the advantages quantitatively, a working consensus based on experience has arisen and is supported by the results of one internal experiment on a real-time multiprocessing system. Simply stated, it is that the added time and thought required to structure a program pay off in better core utilization and improved efficiency which generally are comparable to the effects achieved in unstructured programs by extensive optimization of critical portions. It is also useful to note that even in "critical" programs, a relatively small fraction of the code is really time or core sensitive, and this fraction may not in fact be identifiable during coding. Hence it is probably a better

strategy to use structured coding throughout to begin with. Then, if performance bottlenecks do appear and cannot be resolved otherwise, at most small units code must be hand tailored to remedy the problems. In this way the visibility, manageability, and maintainability advantages of structured coding are largely retained.

Perhaps the most difficult problem to overcome in applying structured coding is the purist syndrome, in which the goal is to write perfectly structured code in every situation. It must be emphasized that structured coding is not an end in itself, but is a means to achieving better, more reliable, more maintainable programs. In some cases (e.g., exiting from a loop when a search is complete, handling interrupt conditions), religious application of the figures allowed by the Guide may produce code which is less readable than that which might contain a GOTO (e.g., to the end of the loop block, or to return from the interrupt handler to a point other than the point of interrupt). Clearly the exceptions must be limited if discipline is to be maintained, but they must be permitted when desirable. To ensure that exceptions are justified, FSD requires documentation and management approval for each one.

### C. Top-Down Development

As defined in Section III-A 3), top-down development is the sequencing of program system development to eliminate or avoid interface problems. This permits development and integration to be carried out in parallel and provides additional advantages such as early availability discussed there.

Top-down development is the most difficult of the four components to introduce, probably because it requires the heaviest involvement and changes of approach on the part of programming managers. Top-down development has profound effects on traditional programming management methodology. While the guidelines sound simple, they require a great deal of careful planning and supervision to carry out thoroughly in practice, even on a small project. The implementation of top-down development, unlike structured coding and DSL's, thus is both a management and a programming problem.

Let us distinguish at this point between what might be called "top-down programming" and true top-down development. While they were originally used interchangeably and the guidelines do not distinguish between them, the two terms are valuable in delineating levels of scope and complexity as use of the methodology increases.

Top-down programming is primarily a single-program-oriented concept. It applies to the development of a "program," typically consisting of one or a few load modules and a number of independently compilable units, which is developed by one or a few programmers. At this level of complexity the problems are primarily ones of program design, and approaches such as "levels of abstraction" [17] and Mills' Expansion Theorem [2] are used. Within this scope of development external

problems and constraints are not as critical as in top-down development, and while management involvement is needed, it need not be so pervasive as in top-down development. Many of FSD's successful projects have used only top-down programming, and the experience gained on them has been most valuable.

Top-down development, on the other hand, is a multiple-program oriented idea. It applies to the development of a "program system," typically consisting of many load modules and perhaps a hundred or more independently compilable units, which is developed by one or more programming departments with five or more people in each. Now the problems expand to those of system architecture, and external problems and constraints become the major ones. The programs in the system are usually interdependent and have a large number of interfaces, perhaps directly but also frequently through shared data sets or communications lines. They may operate in more than one processor concurrently—for example, in a System/7 "front end" and a System/370 "host" or may involve hardware developed especially for the system.

The complexity of such a system makes management involvement in its planning and development essential even when external constraints are minimal. It involves all aspects of the project from its inception to its termination. For example, a proposal for a project to be implemented top-down should differ from one for a conventional (bottom-up) implementation in the proposed manning levels and usage of computer time. Functions must be carefully analyzed during the system design phase to ensure that the practical approach to top-down development (presented in Section III-A 3) above) of minimum code and data dependency is met and a detailed implementation sequence must be planned in accordance with the overall proposed plan and schedule. The design of the system very probably should differ significantly from what it would have been if bottom-up development were to be used. During implementation, progress must be monitored via the D&I to ensure that this sequence is being followed, and their schedules are being met. The early availability of parts of the system must be coordinated with the user if he intends to use these parts for experimentation or production. An entirely different type of test plan must be prepared, for incremental testing over the entire period. Rather than tracking individual components, the manager has the more difficult task of tracking the progress of the system as a whole. In a bottom-up development, the condition of the system is usually not known until the integration phase, when it suddenly becomes a critical item. In top-down work, the condition of the system must always be known, but this knowledge enables the manager to identify problems earlier and to correct them while there is still time to do so.

In typical system development environments such as those in FSD, however, external constraints are the rule rather than the exception. A user will have schedule requirements which must be met. A particular data set must be designed to interface with an existing system.

9. Special hardware may arrive late in a development cycle and may vary from that desired. These are typical of situations not directly under the developers' control which have profound effects on the sequence in which the system is produced. Now the manager's job becomes still more complex in planning and controlling development. Each of these external constraints may force a deviation from what would otherwise be a simple, no-dependency development sequence. Provision may have to be made for testing, documentation, and delivery of products at intermediate points in the overall cycle. This will typically change the schedule from the ideal one, and will probably increase the complexity of the management job. This is especially true on a very large project (several hundred thousand lines of source code or more), since any realistic schedule may well require that major subsystems be developed in parallel and integrated in a nearly conventional fashion (hopefully at an earlier point in time than the end of the project). Top-down development was carried out successfully on a project of 400,000 lines of source code, the largest known to the author to date.

When carried to its fullest extent, top-down development of a large system probably has greater effects on reliability (and thus, indirectly, on productivity) than any other component of the methodology. Even when competent management is fully devoted to its implementation, there are two other problems which can arise and must be planned for. These both relate to the overlapping nature of design, development and integration in a top-down environment.

The first of these concerns the nature of materials documenting the system design to be delivered to and reviewed by the user. Typically, a user receives a program design document at the end of the design phase and must express his concurrence before development proceeds. This is impractical in top-down development because development must proceed in some areas before design is complete in others. To give a user a comparable opportunity, a detailed functional specification is desirable instead. This describes all external aspects of a system, as well as any processing algorithms of concern to a user, but does not address the system's internal design. This type of specification is probably more readily assimilated by typical users, is more meaningful than a design document and should pose no problems in most situations. Where standardized procurement regulations (such as the United States Government Armed Services Procurement Regulations) are in effect, then efforts must be made to seek exceptions (As top-down development becomes more prevalent, then it is hoped that changes to such procedures will directly permit submission of this type of specification.)

The second problem is one of the most severe to be encountered in any of the components and is one of the most difficult to deal with. It has to do with the depth to which a design should be carried before implementation is begun. If a complete, detailed design of an entire system

is done, and implementation of key code in all areas is carried out by the programmers who begin the project, then the work remaining for programmers added later is relatively trivial. In some environments this may be perfectly appropriate and perhaps even desirable; in others it may lead to dissatisfaction and poor morale on the part of the latecomers. It can be avoided by recognizing that design to the same depth in all areas of most systems is totally unnecessary. The initial system design work (the overworked term "architecture" still seems to be appropriate here) should concentrate on specifying all modules to be developed and all intermodule interfaces. Those modules which pose significant schedule, development or performance problems should be identified, and detailed design work and key code writing done only on these. This leaves scope for creativity and originality on the part of the newer programmers, subject obviously to review and concurrence through normal project design control procedures. On some projects, the design of entire support subsystems with interfaces to a main subsystem only through standard, straightforward data sets have been left until late in the project. Note that while this may solve the problems of challenge and morale, it also poses a risk that the difficulty has been underestimated. Thus, here again management is confronted with a difficult decision where an incorrect assessment may be nearly impossible to recover from.

#### D. CPT's

The introduction of CPT's should be a natural outgrowth of top-down development. This is because of the need to complete the system architecture and develop a nucleus before many programmers can work in parallel, and because of the reliance on a DSL, which suggests the use of a small, highly specialized team at the beginning evolving into a larger team later. The use of a smaller group based on a nucleus of experienced people tends to reduce the communications and control problems encountered on a typical project. Use of the other three components of the methodology enhances these advantages through standardization and visibility.

In order for a CPT to function effectively, the chief programmer must be given the time, responsibility, and authority to perform the technical direction of the project. In some environments this poses no problem; in FSD it is sometimes difficult to achieve because of other demands which may be levied upon the chief. In a contract programming environment he may be called upon to perform three distinct types of activities: technical management—the supervision of the development process itself, personnel management—the supervision of the people reporting to him, and contract management—the supervision of the relationships with the customer. The latter in particular can be a very time-consuming function and also is the simplest to secure assistance on. Hence, many FSD CPT's have a program manager who has the primary customer interface responsibility in all non-

technical matters. The chief remains responsible for technical customer interface as well as the other two types of management; in most cases this makes the situation manageable, and if not then additional support can be provided where needed.

The backup programmer role is one that seems to cause people a great deal of difficulty in accepting, probably because there are overtones of "second-best" in the name. Perhaps the name could be improved, but the functions the backup performs are essential and cannot be dispensed with. One of the primary tenets of management is that every manager should identify and train his successor. This is no less true on a CPT and is a major reason for the existence of the backup position. It is also highly desirable for the chief to have a peer with whom he can freely and openly interact, especially in the critical stages of system design. The backup is thus an essential check and balance on the chief. Because of this, it is important that the chief have the right of refusal on a proposed backup; if he feels that an open relationship of mutual trust and respect cannot be achieved, then it is useless to proceed. The requirement that the backup be a peer of the chief also should not be waived, since it is always possible that a backup will be called on to take over the project and must be fully qualified to do so.

One of the limits on a CPT is the scope of a project it can reasonably undertake. It is difficult for a single CPT to get much larger than eight people and still permit the chief and backup to exercise the essential amount of control and supervision. Thus, even at the productivity rates achievable by CPT's it is difficult for a single team to produce much more than perhaps 20,000 lines of code in its first year and 30-40,000 lines after the architecture is complete and the team has grown to full size. Large projects must therefore look to multiple CPT's, which can be implemented in two ways. First, interfaces may be established and independent subsystems may be developed concurrently by several CPT's and then integrated. Second, a single CPT may be established to do architecture and nucleus development for the entire system. It then can spin off subordinate CPT's to complete the development of these subsystems. The latter approach is inherently more appealing, since it carries the precepts of top-down development through intact. It is also more difficult to implement; the experiment under way by the author was not fully successful because equipment being developed concurrently ran into definition problems and prevented true top-down development.

It is difficult to identify problems unique to CPT's which differ from those of top-down development discussed above. Perhaps the most significant one is the claim frequently heard that, "We've had chief programmer teams in place for years—there's nothing new there for us." While it is certainly true that many of the elements of CPT's are not new, the identification of the CPT as a particular form of functional organization using a disciplined, precise methodology suffices to make it unique. In particular, the emphasis on visibility and



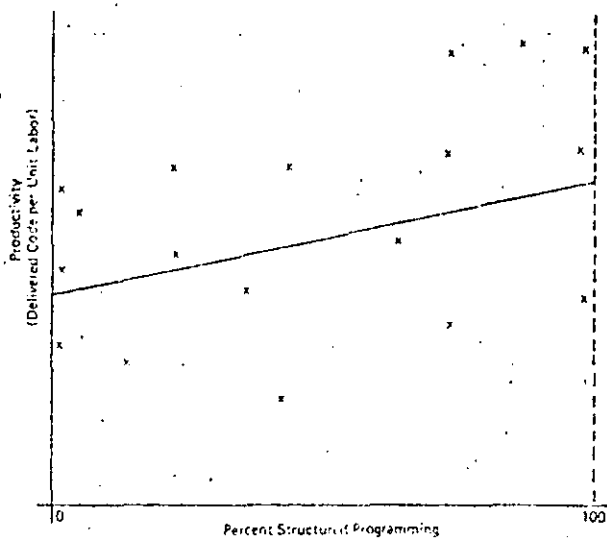


Fig. 4. Productivity trend.

control through management code review, formal structured programming techniques and DSL's differentiate true CPT's from other forms of programming teams [18]. And it is this same set of features which make the CPT approach so valuable in a production programming environment where close control is essential if cost and schedule targets are to be met.

#### V. MEASUREMENT RESULTS

It is not possible, because it would reveal valuable business data, to present significant amounts of quantitative information in this paper. At this time, the results of the measurement program do show substantial improvements in programming productivity where the new technology has been used. Fig. 4 is an idealized version of an actual graph where each point represents a completed FSD project. The horizontal axis records the percentage of structured code in the delivered product, and the vertical axis records the productivity. (The latter includes all effort on the project, including analysis, design, testing, management, support and documentation as well as coding and debugging. It also is based only on delivered code, so that effort used to produce drivers, code written but replaced, etc., tends to reduce the measured productivity.) A weighted least squares fit to the points on the graph shows a better than 1.5 to 1 improvement in the coding rate from projects which use no structured programming to those employing it fully. Since these data were derived from the monthly reports (see Section III-D above), there was no opportunity to test the effects of other factors such as languages or experience, but the results were nevertheless encouraging.

It is also possible, because the data have already been released elsewhere, to make one quantitative comparison between productivity rates experienced using various components of the technology on some of the programming support work which FSD performed for the National Aeronautics and Space Administration's Apollo and Skylab

	Technologies Used	Bytes of New Code (Millions)	Total Effort to Deliver (Man-months)	Productivity (Bytes Per Man-month)
<b>Apollo</b>				
Mission Operations Control	DSL	5.8	3749	1547
Ground Support Simulation	None	2.1	1809	1161
<b>Skylab</b>				
Mission Operations Control	DSL	1.4	1665	841
Ground Support Simulation	DSL SP TDD	4.0	1075	3756

Fig. 5. Productivity comparison.

projects. This comparison is especially significant because the primary identifiable change in approach was the degree to which the new methodology was used; the people, experience level, management and support were all substantially the same in each area. (Other factors may have varied also, but their effects are not clear, and they were not considered important by the participants.) Fig. 5 shows the productivity rates and the components of the technology used. In the Apollo project, a rate of 1161 bytes of new object code per man-month was experienced on the ground support simulation work. (Again, all numbers are based on overall project effort.) This work used none of the components described in this paper. In the directly comparable effort on the Skylab project, a DSL, structured coding, and top-down development were all employed, and a rate of 3756 bytes of new code per man-month was achieved—almost twice as much new code was produced with slightly more than half the effort. It is interesting also to remark that this was achieved on the planned schedule in spite of over 1100 formal changes made during the development of that product, along with cuts in both manpower and computer time. Finally, while the improvement may rest to some extent on the similar work done previously, this was not demonstrated in the parallel mission operations control work. There productivity dropped from 1547 to 841 bytes per man-month on comparable work which in neither case used anything other than a DSL.

In addition to making reliability measurements and determining productivity rates, the measurement activity has served a number of other useful purposes. First, it has built up a substantial data base of information about FSD projects. As new data are submitted, checks are made to ensure its validity, and questionable data are reviewed before being added. The result is an increasingly consistent and useful set of data. Second, it has enabled FSD to begin studies on the value of the components of the methodology. Third, and related, it also permits the study of other factors (e.g., environment, personnel) affecting project activity. Fourth, it is used to assist in reviewing ongoing projects, where the objective data it

contains have proven quite valuable. And fifth, it is used in estimating for proposed projects, where it affords an opportunity to compare the new work against similar work done in the past, and to identify risks which may exist.

## VI. CONCLUSIONS

Reflecting on the benefits of structured programming, one is struck by the fact that the techniques fundamentally are directed toward encouraging programming discipline. Historically, programming has been a very individualistic, undisciplined activity. Thus, introducing discipline (in the form of practices which most programmers recognize as beneficial), yields double rewards—the advantages inherent in the methodology itself, plus those due to better standardization and control.

It should be clear at this point that FSD's experience has been a very positive one. Work remains to be done, particularly in the management of top-down development and the formalization and application of CPT's. Nevertheless, FSD is fully committed to application of the methodology and is continuing to require its use.

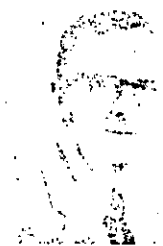
In retrospect, the plan appears to have been a success and could serve as a model for other organizations interested in applying the ideas. The FSD experience shows that this is neither easy nor rapid. It takes substantial time and effort and, most important, commitments and support from management, to equip an organization to apply the methodology.

To summarize, it appears that once a base of tools, standards, and education exists, it is most appropriate to begin with use of DSL's, structured coding and top-down programming. Then, when the people, know-how, and opportunity exist, top-down development should be applied on a few large, complex projects to yield an experienced group of people and the required management techniques. It is likely that one or more of these may also present the opportunity to introduce a CPT. This is essentially the approach that FSD has taken, and it appears to be an excellent way to introduce structured programming.

## REFERENCES

- [1] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, vol. 19, p. 52, May 1973.
- [2] H. D. Mills, "Mathematical foundations for structured programming," IBM Corp., Gaithersburg, Md., Rep. FSC 71-6012, Feb. 1972.
- [3] —, "Chief programmer teams: Principles and procedures," IBM Corp., Gaithersburg, Md., Rep. FSC 71-5108, June 1972.
- [4] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 11, no. 1, pp. 56-73, 1972.

- [5] —, "System quality through structured programming," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41, part 1, Montvale, N. J.: AFIPS Press, 1972, pp. 339-343.
- [6] *Federal Systems Center Structured Programming Guide*, IBM Corp., Gaithersburg, Md., Rep. FSC 72-5075, July 1973 (revised).
- [7] *Improved Technology for Application Development: Management Overview*, IBM Corp., Bethesda, Md., Aug. 1973.
- [8] *FSO-3270 Structured Programming Facility (SPF) General Information Manual*, IBM Corp., Gaithersburg, Md., Form G1120-1638 (available through any IBM branch office).
- [9] F. M. Luppino and R. L. Smith, "Programming support library (PSL) functional requirements," IBM Corp., Gaithersburg, Md., Final Rep., prepared under Contract F30602-74-C-0186 with the U.S. Air Force Rome Air Development Center, Griffiss Air Force Base, Rome, N. Y., July 1974 (release subject to approval of Contracting Officer, P. DeLorenzo).
- [10] *Federal Systems Center Programming Librarian's Guide*, IBM Corp., Gaithersburg, Md., Rep. FSC 72-5074, Apr. 1972.
- [11] P. W. Metzger, *Managing a Programming Project*, Englewood Cliffs, N. J.: Prentice-Hall, 1973.
- [12] R. C. McHenry, *Management Concepts for Top Down Structured Programming*, Gaithersburg, Md.: IBM Corp., Nov. 1972.
- [13] P. W. Metzger and F. R. Bliss, *Programming Project Management Guide*, Gaithersburg, Md.: IBM Corp., Form GA36-0005, July 1974 (available through any IBM branch office).
- [14] *HIPPO-Hierarchical Input-Process-Output Documentation Technique: Audio Education Package*, IBM Corp., Gaithersburg, Md., Form SR20-9413 (available through any IBM branch office).
- [15] M. M. Kessler, *Assembly Language Structured Programming Macros*, Gaithersburg, Md.: IBM Corp., Sept. 1972.
- [16] G. E. Weinburn *et al.*, "Research into the management of computer programming: A transitional analysis of cost estimation techniques," System Development Corp., Santa Monica, Calif., Nov. 1965 (available from the Clearinghouse for Federal Scientific and Technical Information as AD 631 259).
- [17] E. W. Dijkstra, "The structure of the THE multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 341-346, May 1968.
- [18] G. M. Weinburg, *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971.



F. Terry Baker was born in Waterbury, Conn., on January 4, 1935. He received the B.S. degree in mathematics from Yale University, New Haven, Conn., in 1956, and the S.M. degree in applied mathematics from Harvard University, Cambridge, Mass., in 1963.

He has been employed since 1956 by the IBM Corporation in various programming and programming management positions and is currently Manager of the NSDC Development Department, IBM Federal Systems Division, Gaithersburg, Md. He also, on military leave from IBM during the period from 1957 to 1960, served as a lieutenant at the HQ USAF Computer Center in Washington, D.C. Since 1960 he has been active in structured programming, with particular emphasis on its management and on its installation and use in large organizations.

Mr. Baker is a member of Phi Beta Kappa, Sigma Xi, and the Association for Computing Machinery.

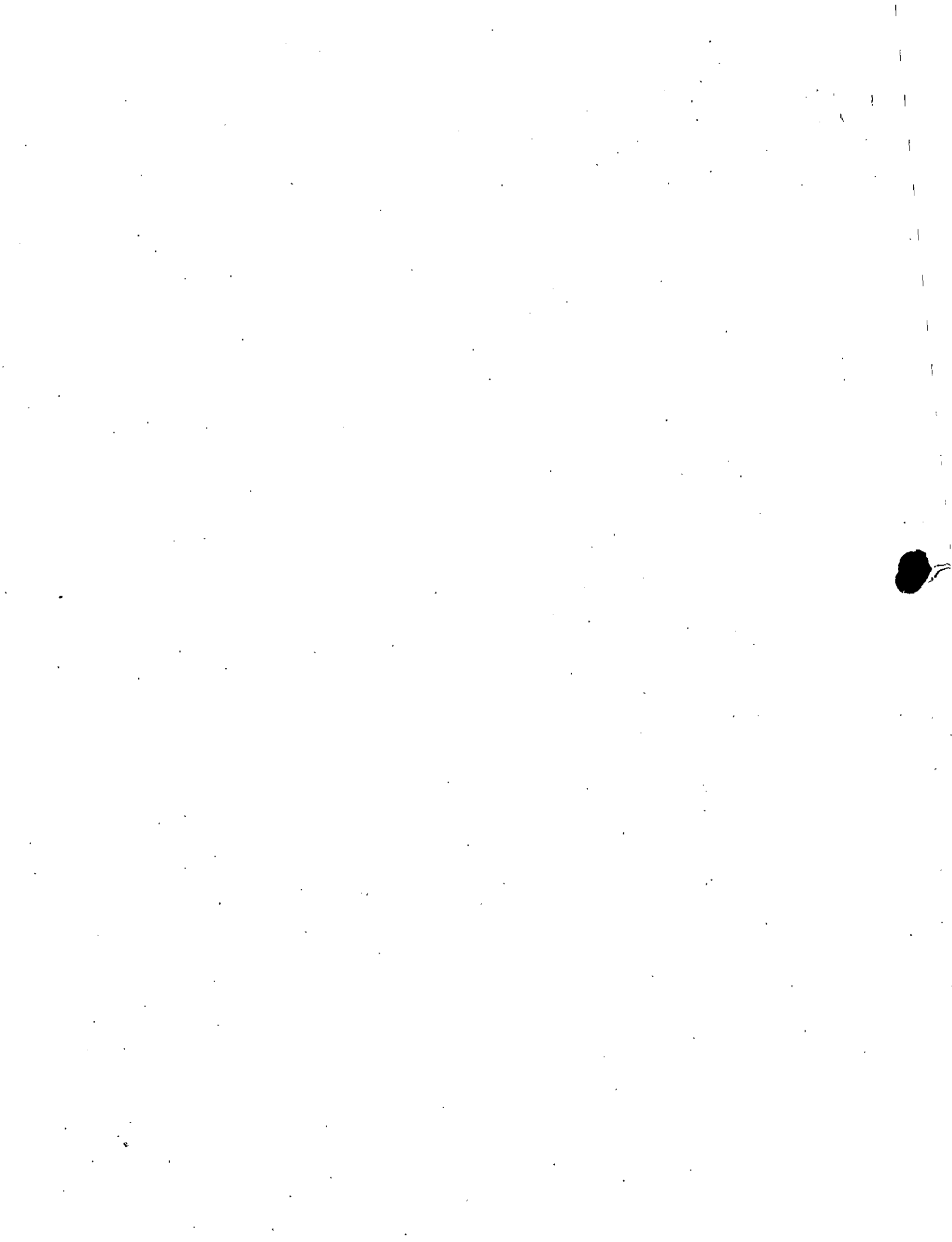


**DIVISION DE EDUCACION CONTINUA  
FACULTAD DE INGENIERIA U.N.A.M.**

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

MODELING SOFTWARE BEHAVIOR IN TERMS OF A FORMAL  
LIFE CYCLE CURVE: IMPLICATIONS FOR  
SOFTWARE MAINTENANCE

DICIEMBRE, 1984



# Modeling Software Behavior in Terms of a Formal Life Cycle Curve: Implications for Software Maintenance

WILLA KAY WIENER-EHRLICH, JAMES R. HAMRICK, MEMBER, IEEE, AND VINCENT F. RUPOLO

**Abstract**—In this paper, a formal model of the software manloading pattern, the Rayleigh model, is described and then applied to four Bankers-Trust Company (BTCo.) new development projects possessing complete life cycle manloading data (maintenance phase included). To fit the Rayleigh curve to a project's manloading scores, (nonlinear) regression was used to obtain least squares estimates of the Rayleigh parameters, which, in turn, were used to generate the Rayleigh manloading curve. For all four projects, deviation from the Rayleigh curve was small and constant throughout the software development phases (i.e., preliminary design through implementation); however, the Rayleigh curve consistently deviated from the actual manloading during system maintenance, underestimating the amount of maneffort expended. Restricting maintenance maneffort to manpower expended on repair of system faults ("corrective" maintenance) resulted in a single Rayleigh curve that could be applied over the entire BTCo. life cycle. Furthermore, this corrective portion of the maintenance effort could be accurately forecasted from the Rayleigh curve fit to software development. Implications of these findings for software management are discussed.

**Index Terms**—Corrective maintenance, development project, empirical, fitted curve, forecasting software maintenance, formal model of software life cycle, projected curve, Rayleigh model, residual score.

## INTRODUCTION

PREVIOUS research on the maneffort loading of medium to large scale software development projects reveals a basic manloading pattern over time: initially, there is a rise in maneffort, followed by a peaking and then a tailing off [13], [14]. The time varying nature of a software project's work profile is based on the following rationale: a software project entails the solution of a fixed number of problems. At each point in time, both the level of skill available for solving problems and the size of the set of unsolved problems available for solution will vary [11], [12]. Since the rate of problem resolution is influenced by both factors, it too will vary over time. Presumably, manpower utilization reflects the rate of problem resolution; hence, the time dependency in the manpower usage curve.

In Fig. 1, the top panel illustrates two mathematical functions that have been proposed as models of the work rate on software development projects: the Rayleigh [14] and the secant squared [13] curves. Observe that for both curves, the manloading rises until it reaches a point of maximum manpower utilization at time equal to  $t_{max}$ . At this point, the manloading

begins to decline at an increasingly rapid rate with time. However, the manpower does not quickly fall to zero. Instead, at  $t_i$ , the inflection point of the declining curve, the manloading begins to decline at a slower and slower rate so that the manloading drops off very slowly. For large scale software development projects (projects with lines of code  $\geq 100,000$ ),  $t_{max}$  is very close to the time of initial operational capability [14]; hence, the rising part of the manloading curve corresponds to the development effort of the project life cycle (that is, the phases planning through implementation), and the falling part of the curve to the operations and maintenance phase (where the principal work is "bug fixing," minor modifications, and enhancements).<sup>1</sup>

Although both the Rayleigh and secant squared (Parr) functions have been shown to approximate the manloading patterns of actual software development projects [1], the Rayleigh curve has been analyzed more extensively. For example, the Rayleigh curve has been applied to several hundred medium to large scale software projects in the areas of logistics, personnel, accounting, and engineering [10], [14]; more recently, it has been shown to apply to smaller scale projects in a commercial environment [21]. However, in much of this empirical work, it is not always clear whether program maintenance was included in the life cycle analysis. Several considerations suggest that software maintenance may have tended not to be included. First, accurate maintenance manloading data is difficult to obtain in organizations that combine the maintenance of several existing software systems into one maintenance project [20] or carry out maintenance in tandem with software development (programmers maintain old systems that they developed while, at the same time, implementing new software). Second, a project's official termination date (and hence the point at which the system receives no further maintenance) is arbitrary and influenced more by economic and managerial factors than by intrinsic forces of the project life cycle [13]. Third, formal accounting procedures for recording the amount of maneffort

<sup>1</sup>Notice that the secant squared and Rayleigh functions are almost identical in the declining right-hand portion of the manloading curve (i.e., during maintenance), but differ in the project's early stages. This is because although both functions have infinite positive tails, only the secant squared curve has an infinite negative tail. (The Rayleigh curve, a discontinuous function, is zero for all negative  $t$ .) Thus, the secant squared model acknowledges preliminary effort done on a project before its official start date,  $t_0$  (due to effort expended on requirements analyses, feasibility studies, and functional specifications), whereas the Rayleigh curve excludes from analysis development work done prior to the start of the project.

Manuscript received August 9, 1982; revised July 19, 1983.

W. K. Wiener-Ehrlich was with Bankers Trust Company, New York, NY 10006. She is now with AT&T Bell Laboratories, Piscataway, NJ.

J. R. Hamrick is with Bankers Trust Company, New York, NY 10006.

V. F. Rupolo is with Dun and Bradstreet, Berkeley Heights, NJ.

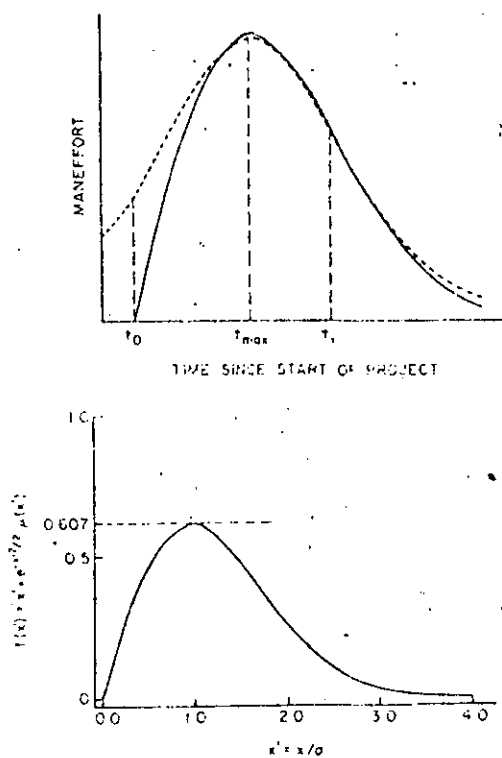


Fig. 1. Formal models of the software life cycle. The top panel presents a comparison of work profiles predicted by the secant squared (dotted line) and Rayleigh (solid line) models. The bottom panel (taken from [21]) presents the rise, peaking, and exponential tailoff behavior of the normalized Rayleigh function

$$f(x') = of(x) = x' * e^{-x'^2/2} / u(x'),$$

$$x' = x/o, \quad o = 1.$$

expanded are typically least likely to be implemented during system maintenance [17].

In summary, although both the Rayleigh and Parr models have been shown to approximate the manloading pattern of software development projects, there is a paucity of empirical data documenting their adequacy in describing the overall life cycle—maintenance phase included. An empirical demonstration would be desirable since, if the models are shown to apply over the entire life cycle, then this would mean that one could predict a software system's maintenance requirements solely on the basis of a manpower curve fit to development manloading. This is an extremely powerful implication of the formal life cycle approach to software behavior, and one that is of considerable practical importance, especially in light of the fact that as much as 60 percent of the work done on a system is software maintenance [14].

#### MODELING SOFTWARE BEHAVIOR IN TERMS OF THE RAYLEIGH LIFE CYCLE CURVE—AN EMPIRICAL EXAMPLE

In the present study, the accuracy of one formal model of the software life cycle, the Rayleigh curve, was analyzed for software projects in a commercial programming environment. Four new development projects possessing complete manloading data (systems maintenance included) were selected for analysis. All projects were medium-sized development projects (delivered lines of code <80 000; see Table I) implemented at the Bankers Trust Company (BTCo.) during the period from

1976 to 1980.<sup>2</sup> The Rayleigh function was selected as the formal life cycle model since previous research [21] had indicated that it could be applied to BTCo. project's development efforts.

#### Applying the Rayleigh Curve to Software Life Cycle

To apply the one parameter Rayleigh function

$$y = f(x) = \frac{x}{o^2} * e^{-(x^2/2o^2)} \quad 0 < x \leq \infty$$

$$= 0 \quad -\infty \leq x \leq 0 \quad (1)$$

(see Fig. 1, bottom panel) to the software life cycle, (1) must be modified as follows: elapsed time from the start of the project  $t$  will be substituted for  $x$ ; time of peak manpower  $t_{max}$  will be substituted for the Rayleigh parameter  $o$ , and a second parameter,  $K$ , will be introduced into the equation. This parameter adjusts the manloading at each value of  $t$  by the constant  $K$  to take into account differences in project size due to the total maneffort. This second parameter represents the total cumulative maneffort utilized by the end of the project and is equal to the area under the Rayleigh curve. The two-parameter Rayleigh function can be represented as follows:

$$y = f(t) = K/t_{max}^2 * t * e^{-(t^2/2t_{max}^2)} \quad (2)$$

If (2) is rewritten as:

$$y = f(t) = 2at * Ke^{-at^2},$$

$$a = \frac{1}{2t_{max}^2} \quad (3)$$

then it can be observed that the Rayleigh function is made up of two time varying components. One component,  $2at$ , increases linearly over time, while the second component,  $Ke^{-at^2}$ , is an exponentially decreasing function over time. Presumably, the maneffort loading at  $t$  reflects the number of problems that can be solved at  $t$ ; therefore, (3) states that the number of problems solved at  $t$  is equal to the number of problems available for solution at  $t$ ,  $Ke^{-at^2}$ , times the probability that a problem will be solved at  $t$ ,  $2at$ .

Fig. 2 illustrates the effects of the  $a$  and  $K$  parameters on the Rayleigh curve. The top panel indicates that the effect of the  $a$  parameter (scale parameter) is to compress or stretch out the manloading distribution and hence the duration of the project. Large  $a$  values (small  $t_{max}$  values) result in sharply peaked manloading distributions with rapid manpower buildup and phase-out (projects of short duration), while small  $a$  values (large

<sup>2</sup>Unfortunately, complete life cycle manloading curves were not available for other BTCo. new development projects. This is because in the BTCo. environment a development project terminates once the system becomes operational. Its maintenance phase is assigned to a different project team (typically responsible for the concurrent maintenance of many similar systems) or else its maintenance is embedded within the subsequent phase of a multiphase application effort. Consequently, maintenance manloading tended to be unavailable for BTCo. development projects. For four development projects (BACC12, CASH1, ACCTANLY, and ACCESS1), however, their maintenance teams serviced only the single system's requests. Therefore, for these four projects, complete manloading distributions could be obtained that included the maintenance phase of the project life cycle.

### 3 SUMMARY PROJECT CHARACTERISTICS

Project Name	System Size (in Procedural Division Lines of Code)	Total Project Effort <sup>a</sup> (including Post- Implementation, in Man-Months) <sup>b</sup>	Language Used	Formal Systems Development Methodology (SDM)	Project Purpose
BACC12	79,534 LOC	193.66 MM	BASIC, MACRO	BTCO. SUR	Provides processing for management and regulatory reporting.
CASHC1	34,677 LOC	153.71 MM	FORTRAN, MACRO	BTCO. SDM	Processes the Balance Reporting and Money Transfer Reporting products
ACCTARLY	50,131 LOC	193.5 MM	COBOL	BTCO. SDM	Provides processing for reporting of financial (Account Analysis Statements) and management (relative profitability of BTCO. wholesale customers) information
ACCESS1	21,300 LOC	143.17 MM	COROL	BTCO. SDM	Retail banking application designed to provide inquiry and hold capabilities for checking and savings accounts.

<sup>a</sup>Effort is defined as cumulative number of BTCO. staff and vendor personnel on the project for each month of the project's duration (1 manmonth = 125 manhours).

<sup>b</sup>Project manloading data were obtained from BTCO.'s financial reporting system. Data were validated by reconciling monthly manpower estimates against mancounts given in actual manpower expenditure reports.

$t_{max}$  values) result in more gradual distributions ("stretched out" projects) of longer duration. The  $K$  parameter (bottom panel) changes only the total effort under the curve, but not the shape (degree of peakedness).

#### Analytical Techniques

Two basic techniques were used for the Rayleigh curve fitting. First, regression analysis<sup>3</sup> was used to obtain least squares estimates of the Rayleigh model's parameters,  $K$  and  $a$ , which, in turn, were used to generate a project's Rayleigh manloading curve. Second, the differences between the observed and fitted Rayleigh manloading scores (the residual scores) were analyzed to identify regions of poor model fit. These differences are important because they reveal where in the software life cycle the Rayleigh curve is doing poorly. Thus, by analyzing the residual scores over time, one can determine whether there is systematic departure from the fitted Rayleigh equation.

<sup>3</sup>Equation (3) is nonlinear in its parameters; consequently, nonlinear least squares regression, Marquardt procedure, was the regression technique used for the curve fitting. The Marquardt procedure is an iterative parameter estimation technique (i.e., it keeps revising the parameter estimates until it converges to the final least squares estimates). Therefore, initial values for the Rayleigh parameters needed to be specified [4]. To obtain these initial starting values, the Rayleigh function was evaluated for every parameter combination in a region of possible parameter values defined by varying  $K$  from 20 to 400 manmonths and varying  $a$  from 0.0555 to 0.0005 (corresponding to  $t_{max} = 3$  to 30 months). These particular  $K$ ,  $t_{max}$  values were selected on the basis of prior research which indicated that BTCO. development projects tended to exhibit  $K$ ,  $t_{max}$  values that fell within this region (see Table II in [21]). A reduced set of parameter combinations in the parameter space surrounding the point with the smallest error sum of squares was then examined and the pair with the minimum error sum of squares was used to start off the iterative estimation.

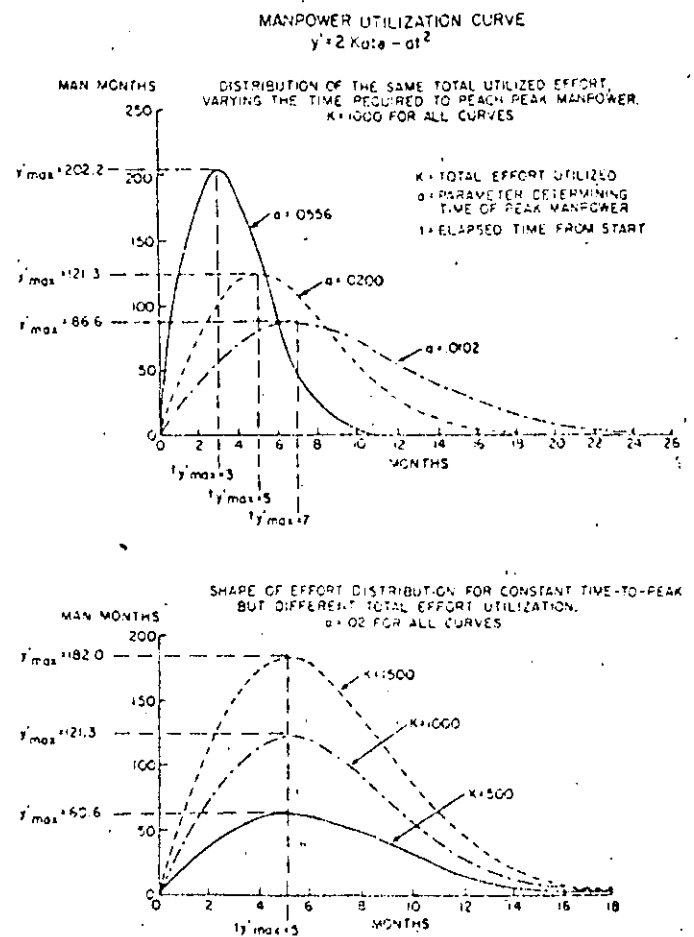


Fig. 2. Effect of changing parameter values ( $K$  and  $a$ ) on Rayleigh life cycle curve. The top panel illustrates the effect of varying  $a$  given a constant  $K$ . The bottom panel shows the effect of varying  $K$  given a constant  $a$ . (Taken from [12].)

	Portion of Life Cycle Analyzed											
	Entire Life Cycle				Systems Development Only				Systems Maintenance Only			
	$k^d$	$t_{max}^b$	$k^d$	$t_{max}^b$	$k^d$	$t_{max}^b$	$k^d$	$t_{max}^b$	$k^d$	$t_{max}^b$	$k^d$	$t_{max}^b$
Point	95% Confid. Intvl.	Point	95% Confid. Intvl.	Point	95% Confid. Intvl.	Point	95% Confid. Intvl.	Point	95% Confid. Intvl.	Point	95% Confid. Intvl.	
BACC12	163.37	170.52	8.37	7.96	180.96	160.35	8.20	7.69	83.35	72.53	14.40	12.48
		194.23		8.84		195.57		8.82		94.17		17.62
CASHC1 <sup>c</sup>	137.24	113.71	5.91	5.29	133.83	109.28	5.61	4.87	100.63	83.60	10.96	8.94
		150.94		6.82		158.38		6.83		117.70		15.58
ACCTANLY <sup>d</sup>	166.91	170.74	8.11	7.62	192.83	176.57	8.34	7.79	165.0	365.49	31.42	13.90
		203.09		8.70		209.09		9.04		595.49		-17.75
ACCESS1 <sup>e</sup>	81.58	65.01	4.54	4.02	82.73	62.93	4.48	3.75	84.59	72.03	15.98	14.50
		90.75		5.32		102.53		5.90		97.14		18.03

<sup>a</sup>Total cumulative effort (in manmonths units) expended on project.

<sup>b</sup>Time of peak manloading (in months) since start of project.

<sup>c</sup>Time periods 11 and 12 excluded from maintenance only analysis due to inaccurate manloading counts.

<sup>d</sup>Fitted parameters in maintenance only least square solution are highly negatively correlated (-0.998) with large 95 percent confidence regions.

<sup>e</sup>Time periods 9 and 10 excluded from maintenance only analysis due to inaccurate manloading counts.

### Results of Rayleigh Curve Fitting

*Estimates of Rayleigh Parameters:* Table II, columns 1-4, presents the least squares estimates of the Rayleigh parameters. (Observe that the  $t_{max}$  parameter is presented in place of the  $a$  parameter.) Both point and interval estimates (the two extreme points in the parameter's 95 percent confidence interval) are included. The point estimate refers to the single "best" estimate of a model's parameter while the interval estimate indicates a range of likely values within which the parameter lies. The interval estimate is important because it indicates the degree of variation among "likely" parameter values. For all four projects, the range of  $K$ ,  $t_{max}$  values considered by the data as not unreasonable for the true values of  $K$ ,  $t_{max}$  are fairly tight, implying good least squares solutions.

The best fitting Rayleigh curves (generated by substituting the fitted parameters back into (3) for the time periods analyzed) together with the actual manloading distributions are presented in Fig. 3.<sup>4</sup> Notice that in all four fitted curves, the empirically determined time of initial operational capability  $t_d$  (equal to 17 for BACC12, 10 for CASHC1, 16 for ACCTANLY, and 8 for ACCESS1) does not coincide with the fitted time of peak manloading,  $t_{max}$  (equal to 8.37 for BACC12, 5.91 for CASHC1, 8.11 for ACCTANLY, 4.54 for ACCESS1). Rather,  $t_d$  is approximately twice  $t_{max}$  ( $t_d \approx 2t_{max}$ ) and therefore occurs near

the end of the Rayleigh curve, past the point of inflection ( $t_i = 1.7t_{max}$ ). Putnam [15] has indicated that for medium-sized systems (18 000-70 000 lines of code),  $t_{max}$  is approximately midway between the start of the project  $t_0$  and  $t_d$ ; hence, these Rayleigh curves indicate that BTCO development projects peak in the manner as other medium-sized projects reported in the literature.

*Examination of Residuals:* Inspection of Fig. 3 reveals two basic residual score patterns. (In the present context, residual scores refer to the difference between actual and Rayleigh manloading values at time  $t$ .) First, except for the initial underestimation at  $t = 1$  (a finding also reported by [21]<sup>5</sup>), deviation from the Rayleigh model is small and constant throughout the phases preliminary design through implementa-

<sup>5</sup>Wiener-Ehrlich *et al.* [21] interpreted the Rayleigh model's initial underestimation of project manloading (at  $t = 1$ ) as indicative of failure to identify the "true" start of the project. They suggested introducing a location (origin) parameter into the Rayleigh equation in order to statistically establish the point of zero manloading and hence the project's start date. If the location parameter,  $\gamma$ , is incorporated into (3) then the resulting equation

$$y = f(t) = 2Ka * (t - \gamma) * e^{-a(t-\gamma)^2} \quad (4)$$

is equivalent to the Weibull function with a shape parameter equal to 2 [6], [8]. (Parameter  $K$  adjusts the manloading at each value of  $t$  by a constant to take into account differences among projects in terms of the total maneffort.) The results of fitting the Weibull function to project data indicated that although the effect of the parameter was to reduce the error sum of squares (a result also reported by [1]), it resulted in an overparameterized model for some data sets. Apparently, for these data sets, the data were inadequate to allow estimation of three parameters. Therefore, the parameter was eliminated from (4), and (3) fit directly to the data.

<sup>4</sup>Two of the BTCO life cycle phases, feasibility and survey/analysis, were omitted from the life cycle analysis. This was because several investigators [16] had indicated that these initial phases are not part of a project's formal development. Consequently, the start of the project  $t_0$  was set equal to the beginning of preliminary design.



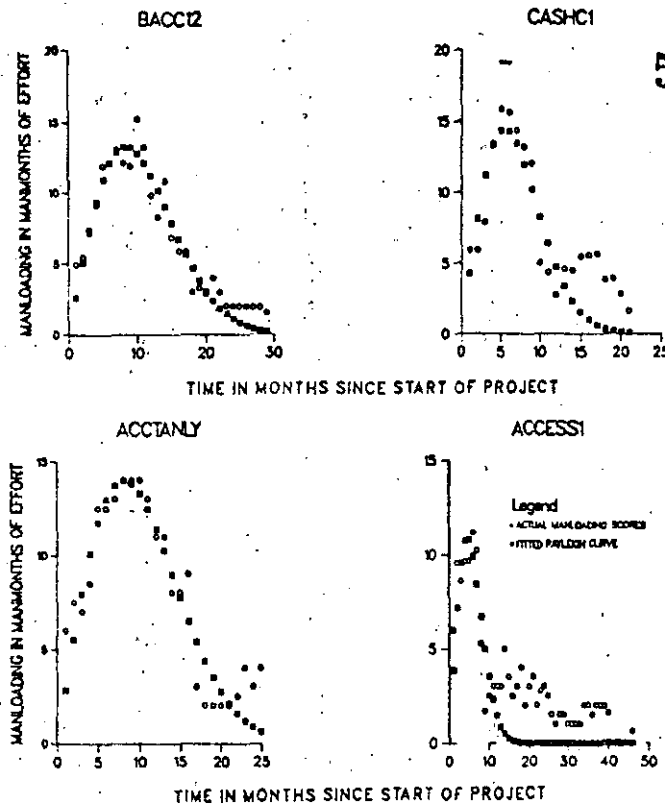


Fig. 3. Actual manloading scores together with best fitting Rayleigh curves for four BTCO. new development projects. In this analysis, Rayleigh curves were fit to projects' manpower scores over the entire life cycle (i.e., phases preliminary design through maintenance). For BACC12 (top left), the life cycle extends from  $t = 1$  to  $t = 29$  (development:  $t = 1-17$ , maintenance:  $t = 18-29$ ); for CASHC1 (top right), from  $t = 1$  to  $t = 21$  (development  $t = 1-10$ , maintenance:  $t = 11-21$ ); for ACCTANLY (bottom left), from  $t = 1$  to  $t = 25$  (development:  $t = 1-16$  development, maintenance:  $t = 17-25$ ); and for ACCESS1 (bottom right), from  $t = 1$  to  $t = 46$  (development:  $t = 1-8$ , maintenance:  $t = 9-46$ ).

tion. Second, the Rayleigh curve consistently deviates from the actual manloading at the end of the project life cycle, underestimating the maneffort expended during maintenance. For two projects, BACC12 and CASHC1, the maintenance manloading rises and then falls in a Rayleigh-like fashion, while for one project, ACCESS1, the maintenance manning appears to follow two small Rayleigh-like curves. Since the enhancements that occur during maintenance by their very nature necessarily require design and development (they give the system capabilities not called for in the original functional specifications), it is not surprising that BTCO. maintenance manning tends to follow its own separate Rayleigh curve.

#### Forecasting Software Maintenance from the Development Derived Rayleigh Curve

By the time the BTCO. operations phase is reached, approximately 86 percent of the total life cycle effort has been expended on developing the software system.<sup>6</sup> Since, by this point, a significant number of manloading values exist for fitting a Rayleigh curve, one should be able to use the Rayleigh curve fit to a system's development effort to predict what its

<sup>6</sup>This value was obtained by integrating the Rayleigh manloading curve (3) from  $t = 0$  to  $t = t_d$ ,  $t_d = 2t_{max}$ , which gives the cumulative manpower utilization at  $t_d$ .

manloading requirements will be during maintenance. A condition for forecasting maintenance is that the development and maintenance manloading values be described by the same Rayleigh function (i.e., that both sets of scores lie on the same Rayleigh curve). To determine whether this is true for the BTCO. environment, one needs to compare the Rayleigh curves derived from BTCO. projects' maintenance phases to the Rayleigh curves derived from their development efforts. If the two sets of Rayleigh parameters,  $K$  and  $a$  ( $t_{max}$ ), tend to be similar, then this would constitute strong empirical evidence for the ability to forecast maintenance behavior from a development derived Rayleigh curve.

The results of fitting separate Rayleigh curves to BTCO. projects' development and maintenance efforts<sup>7</sup> are presented in Table II, columns 5-12. For the development manloading, the parameter estimates appear to be reasonable estimates since, for all four projects, the probable parameter values cluster tightly about the point estimates. In terms of the maintenance manloading, the least squares solutions (columns 9-12) appear to be desirable save for one project, ACCTANLY, due to the  $a$  parameter's large standard error and its high correlation with the  $K$  parameter. Table I indicates that for all four projects, the development and maintenance parameters are highly discrepant (compare BACC12's Rayleigh parameters  $K = 180.96$  man-months,  $t_{max} = 8.2$  months based on development data with the parameters  $K = 83.35$  man-months,  $t_{max} = 14.4$  months obtained from maintenance data), suggesting that the two sets of manloading do not come from the same Rayleigh curve. Hence, one cannot use Rayleigh curves derived from BTCO. projects' development efforts to forecast future manloading requirements during system maintenance.

The Rayleigh curves derived from projects' development and maintenance efforts projected over the entire life cycle, together with the actual manloading scores, are presented in Figs. 4 and 5. (In Fig. 4, projecting maneffort loadings into the maintenance phase is accomplished by using the development function parameters to generate manloading values for subsequent time periods. In Fig. 5, Rayleigh manloading values for project development were obtained by using the maintenance function parameters to generate maneffort values for earlier time periods.) Observe that in Fig. 4, although the projects' development manloading closely approximates the fitted Rayleigh curves,<sup>8</sup> the maintenance manloading deviates markedly from the projected Rayleigh pattern. In Fig. 5, Rayleigh func-

<sup>7</sup>In fitting Rayleigh curves to projects' maintenance manloading data, the time scale was properly adjusted with respect to  $t_0$ . Thus, the maintenance time periods'  $t$  values were set equal to the number of months following the start of preliminary design.

<sup>8</sup>In order to statistically evaluate the Rayleigh curve's goodness-of-fit, a chi-square test [18] was performed on each project's actual versus fitted manloading scores. The idea behind this is to compare the observed manloading for a given time period to the number that would be expected on the basis of the best fitting Rayleigh curve. The comparison can be made in such a way that the resulting test statistic has an approximate chi-square distribution, with degrees of freedom  $df$  equal to  $k - 1 - t$  [ $k$  is the number of cells (time periods) and  $t$  is the number of parameters estimated]. The results of the chi-square test applied to development only manloading were all highly nonsignificant and are as follows: BACC12:  $\chi^2 = 2.82$ , pooled  $df = 13$ ,  $p < 0.995$ ; CASHC1:  $\chi^2 = 3.05$ , pooled  $df = 6$ ,  $p < 0.9$ ; ACCTANLY:  $\chi^2 = 4.8$  pooled  $df = 12$ ,  $p < 0.975$ ; ACCESS1:  $\chi^2 = 2.85$ , pooled  $df = 4$ ,  $p < 0.75$ .

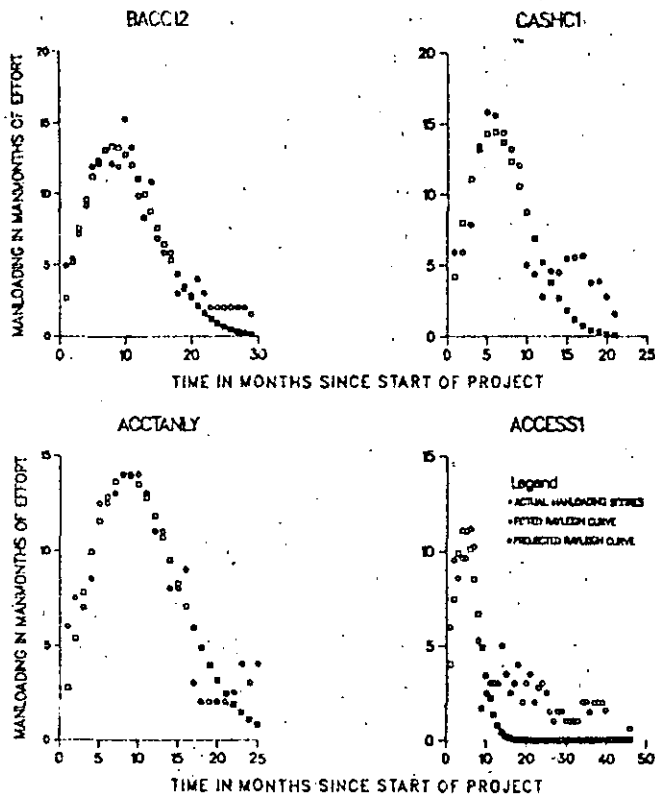


Fig. 4. Actual manloading scores together with best fitting Rayleigh curves for four BCo. new development projects. In this analysis, Rayleigh curves were fit to projects' development efforts only (i.e., phase preliminary design through implementation), and then projected into maintenance. Fitted versus projected Rayleigh curves are denoted by different symbols.

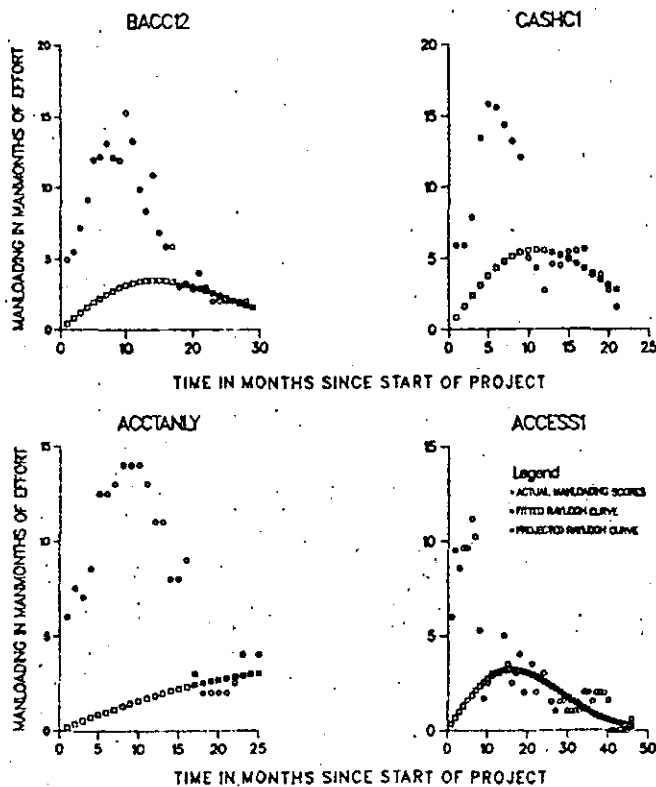


Fig. 5. Actual manloading scores together with best fitting Rayleigh curves for four BCo. new development projects. In this analysis, Rayleigh curves were fit to projects' maintenance efforts only, and then projected back into development. Fitted versus projected Rayleigh curves are represented by different symbols.

6

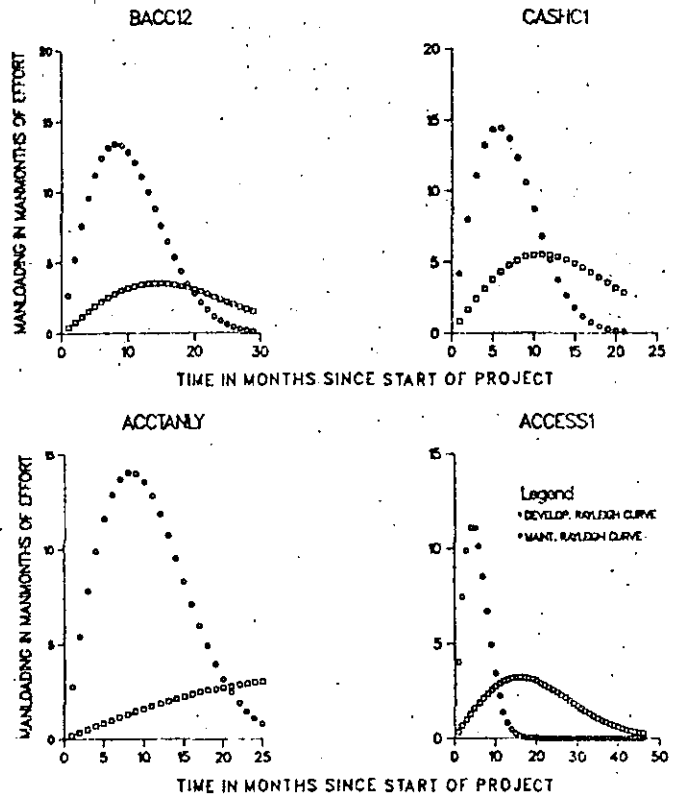


Fig. 6. Comparison of Rayleigh curves derived from BCo. projects' development versus maintenance efforts, projected over the entire life cycle.

tions also characterize software maintenance,<sup>9</sup> but these functions differ from the development Rayleigh curves and hence do not describe software development.

Fig. 6 presents the projects' development and maintenance derived Rayleigh curves projected over the entire life cycle. Observe that for all four BCo. projects, the point of peak manloading occurs later in the maintenance curves, resulting in distributions with more gradual buildup and phaseout than the steeper, more sharply peaked development curves. Since the effect of increasing  $t_{max}$  is to increase the manloading at the end of the project (see Fig. 2, top panel), large  $t_{max}$  values are required for the maintenance curves in order to depict the large maneffort expenditures that occur during systems maintenance. What these curves signify, then, is that at BCo. more effort is being expended on software maintenance than is predicted on the basis of maneffort expended during software development. In summary, BCo. development derived Rayleigh curves are poor predictors of maintenance staffing since they significantly underestimate the amount of manloading required.

#### IMPLICATIONS FOR SOFTWARE MAINTENANCE

In order to understand the implication of these findings for systems maintenance, first consider the types of activities that are performed during this phase of the software life cycle.

<sup>9</sup>The results of the chi-square test applied to maintenance only manloading were also all nonsignificant and are as follows: BACC12:  $\chi^2 = 0.25$ , pooled  $df = 2$ ,  $p < 0.9$ ; CASHC1:  $\chi^2 = 1.27$ , pooled  $df = 3$ ,  $p < 0.75$ ; ACCTANLY:  $\chi^2 = 1.05$ , pooled  $df = 1$ ,  $p < 0.5$ ; ACCESS1:  $\chi^2 = 5.0$ , pooled  $df = 8$ ,  $p < 0.9$ .

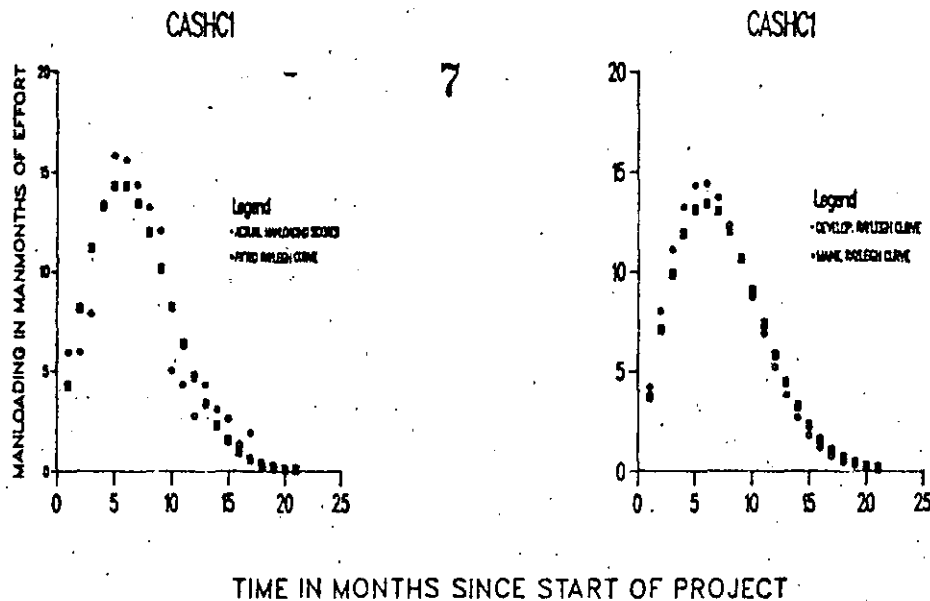


Fig. 7. The Rayleigh curve as a model of the BCo. life cycle, given a revised definition of software maintenance. The left-hand panel presents CASHCI's actual manloading scores (maintenance manloading restricted to corrective maintenance activities only) together with the best fitting Rayleigh curve. A comparison between CASHCI's development and maintenance derived Rayleigh curves projected over the entire life cycle is presented in the right-hand panel.

During maintenance, a system typically engages in several different activities designed to restore and retain operational effectiveness. These include correction of system faults, performance improvements, additions and modifications to existing functionality, and support of new software and hardware. Only the first of these activities, identification of original design defects and error correction (i.e., "corrective" maintenance [7]), are intrinsic to the programming task. In contrast, the latter activities (termed "perfective" and "adaptive" maintenance) are the result of factors external to the programming process that are difficult to predict: for example, the maintenance group's initiative [19], availability of new hardware devices and systems software [5], tight development schedules [9], and changes in the business and user environments [3].

The major result from the present study—that more maneffort is expended on maintenance than was predicted in terms of a development derived Rayleigh curve—suggests that for the purpose of life cycle forecasting, only some, not all maintenance activities be included in the maintenance phase of the software life cycle. For systems that peak in the manner of BCo. systems (i.e., where  $t_{max} = 1/2 t_d$ ), it would seem that only corrective maintenance should be included. This is because, first, corrective maintenance represents the culmination of the software development process and, hence, its maneffort should be predicted from system development. Second, recall that by the time BCo. projects enter the maintenance phase  $t_d$ , the inflection point of the Rayleigh curve has passed and the manpower is leveling off at an increasingly reduced rate over time (see Fig. 1). Since the manpower is assumed to be proportional to the rate of problem resolution, the long tail of the Rayleigh curve implies that the number of problems solved is small and relatively constant. Detection and repair of software system faults (where the fault is due to faulty implementation, weakness in design, or incorrect functional specifications) would also seem to exhibit a pattern of leveling off after the initial

startup period. Therefore, only corrective maintenance effort should fall along the same Rayleigh curve that applies to system development.

For one project, CASHCI, maintenance data was available concerning the amount of maneffort expended (per month) on "bug fixing," modifications to conform to functional specifications and new enhancements.<sup>10</sup> The first two categories exemplify repair of system faults; consequently, their joint maneffort should correspond to corrective maintenance. Fig. 7, left-hand panel, presents the actual manloading distribution (with the maintenance manloading restricted to only these two maintenance activities), together with the fitted Rayleigh curve. The chi-square test was highly nonsignificant ( $X^2 = 5.33$ , pooled  $df = 9$ ,  $p \leq 0.9$ ), indicating that the actual manloading values conform to the Rayleigh model. (Observe that there is no tendency for the Rayleigh model to underestimate manloading during maintenance.) The best fitting Rayleigh curves derived from CASHCI's development and maintenance efforts, projected over the entire life cycle, appear in the right-hand panel of Fig. 7. The fitted curves are very similar to each other and to the overall Rayleigh curve in the first panel, due to their almost identical Rayleigh parameter estimates ( $K = 130.03$  manmonths,  $t_{max} = 5.48$  months for both development and maintenance;  $K = 133.83$  manmonths,  $t_{max} = 5.61$  months for development;  $K = 131.04$  manmonths,  $t_{max} = 5.93$  months for maintenance). This result indicates that the development and maintenance manloading come from the same Rayleigh distribution; thus, a single Rayleigh curve applies over the entire software life cycle. Hence, we could have used CASHCI's development derived Rayleigh curve to predict its maintenance manpower staffing. Thus, given systems that peak in the same manner as BCo. projects, if one considers

<sup>10</sup>The authors would like to acknowledge the assistance of J. Acca in obtaining these data.

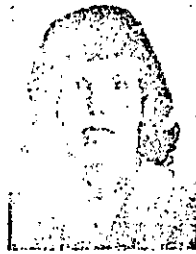
only the maneffort expended on correcting original design and specification faults, plus bugs and errors, then this maneffort can be predicted from development maneffort using a Rayleigh life cycle curve.

Although this finding comes from an analysis of only one project, the results suggest that at BTCo. the Rayleigh curve may apply over the entire software life cycle, when given a restricted definition of software maintenance. This may be an important implication for management since it implies that one does not have to know the properties of the developed source code (e.g., size, structure, syntax) nor the software quality, to estimate a system's corrective maintenance requirements. Instead, one has to know only the project staffing during software development to make the maintenance forecast. The empirical work also has implications for managing software enhancements that extend an existing system's capabilities. Rather than lumping all software enhancements together with system fault correction under a single maintenance category, enhancements should be organized as separate software projects. This is because, when included in the maintenance phase of the software life cycle, enhancements result in manloading curves that do not exhibit a Rayleigh pattern throughout the entire life cycle. However, when managed as separate projects, system enhancements do follow the Rayleigh curve [21]. Consequently, there is greater opportunity for software planning and control under this latter software management strategy.

#### REFERENCES

- [1] V. R. Basili and J. Beane, "Can the Parr curve help with manpower distribution and resource estimation problems?" *J. Syst. Software*, vol. 2, pp. 59-69, 1981.
- [2] V. R. Basili and M. V. Zelkowitz, "Analyzing medium-scale software development," in *Proc. 3rd Int. Conf. Software Eng.*, 1978, pp. 116-123.
- [3] L. A. Belady and M. M. Lehman, "A model of large program development," in *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, Ed. New York: Comput. Soc. Press, 1980, pp. 65-92.
- [4] N. R. Draper and H. Smith, *Applied Regression Analysis*, 2nd ed. New York: Wiley, 1981.
- [5] R. L. Glass and R. A. Noiseux, *Software Maintenance Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [6] F. M. Gryna, "Basic statistical methods," in *Quality Control Handbook*, 3rd ed., J. M. Juran, F. M. Gryna, and R. S. Bingham, Eds. New York: McGraw-Hill, 1974.
- [7] E. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 466-471, 1978.
- [8] N. R. Mann, R. E. Schafer, and N. D. Singpurwalla, *Methods for Statistical Analysis of Reliability and Life Data*. New York: Wiley, 1974.
- [9] C. L. McClure, *Managing Software Development and Maintenance*. New York: Van Nostrand Reinhold, 1981.
- [10] W. Myers, "A statistical approach to scheduling software development," *Computer*, pp. 23-35, 1978.
- [11] P. V. Norden, "Useful tools for project management," in *Management of Production*, M. K. Starr, Ed. Baltimore, MD: Penguin, 1970.
- [12] —, "Project life cycle modeling: Background and application of the life cycle curves," presented at Software Life Cycle Management Workshop, Airlie, VA, Aug. 1977.
- [13] F. N. Parr, "An alternative to the Rayleigh curve model for software development effort," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 291-296, May 1980.
- [14] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 345-361, 1978.

- [15] —, "Special topics," in *Tutorial Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers*, L. H. Putnam, Ed. New York: Comput. Soc. Press, 1980.
- [16] L. H. Putnam and A. Fitzsimmons, "Estimating software costs, Part I," *Datamation*, pp. 189-198, Sept. 1979.
- [17] V. Rupolo, "Application software maintenance at the Bankers Trust Company," Bankers Trust Company, Internal Memo, Nov. 1981.
- [18] S. S. Shapiro, *How to Test for Normality and Other Distributional Assumptions*, vol. 3. Milwaukee, WI: Amer. Soc. Quality Contr., 1980.
- [19] E. B. Swanson, "The dimension of maintenance," in *Proc. 2nd Int. Conf. Software Eng.*, 1976, pp. 492-497.
- [20] W. K. Wiener-Ehrlich, "Increasing systems development productivity," Bankers Trust Company, unpublished, Oct. 1980.
- [21] W. K. Wiener-Ehrlich, J. Hamrick, and V. Rupolo, "Applicability of the Rayleigh model to three different types of software projects," in *Proc. 23rd IEEE Comput. Soc. Int. Conf.*, 1981, pp. 128-148.



Willa Kay Wiener-Ehrlich received the B.A. degree in psychology from the State University of New York at Stony Brook in 1970 and the Ph.D. degree in psychology from the University of Minnesota, Minneapolis, in 1974.

From 1974 to 1976 she was a Research Fellow at the Brown University Center for Human Learning, where she investigated mathematical and programming models of cognitive processes. From 1977 to 1979 she was an Instructor at the Brown University Medical School, where she provided mathematical, statistical, and programming support to several research projects. From 1980 to 1983 she served as a Senior Technical Consultant for improving software productivity at Bankers Trust Company. She is currently a member of Technical Staff, AT&T Bell Laboratories, Piscataway, NJ. Her main research interests are in software productivity measurement and evaluation, software cost estimation, project planning, and control.



James R. Hamrick (M'81) received the B.S. degree in mathematics with a minor in physics from Utah State University Logan, 1961.

His 22 years of experience has included senior management responsibilities for quality assurance, human resource development and training, data center facilities, systems programming, business counseling, and strategic planning. He is currently Vice President, Strategic Planning and Consulting, in the Trust and Securities Systems Development Department of Bankers Trust Company.

Mr. Hamrick is a member of the ACM, the North American Society for Corporate Planning, the Society for Strategic and Long Range Planning, the Association for Systems Management, and the American Management Association.



Vincent F. Rupolo received the B.A. in mathematics from Providence College, Providence, RI, and the M.S. in computer science from Fairleigh Dickinson University, Rutherford, NJ.

He is Director of Project Management for Dun and Bradstreet, Berkeley Heights, NJ. His responsibilities include the selection, evaluation, and support of software tools and techniques to improve systems development productivity and the development of standards and guidelines for their use. His more than 19 years of experience include work for the General Foods Corporation, specifically in the design of promotion information systems. He has an extensive background as a Data Processing Consultant in the areas of management consulting and the management of information systems development.

Mr. Rupolo is a member of the ACM.

- [2] "FASP management summary," U.S. Naval Air Development Center, Warminster, PA, Apr. 1979; "FASP software production and maintenance methodology," U.S. Naval Air Development Center, Warminster, PA, July 1979; and *FASP Handbook*, U.S. Naval Air Development Center, Warminster, PA, Dec. 1979.
- [3] F. L. Bauer, "Software engineering," in *Proc. IFIPS Congr.*, 1971, pp. 1-267, 1-274.
- [4] "Support software planning study," Softech, Inc. Contract N62269-74-C-0269, U.S. Naval Air Development Center, Warminster, PA, Mar. 1974.
- [5] J. H. Morrissey and L. S.-Y. Wu, "Software engineering... An economic perspective," in *Proc. 4th Int. Conf. Software Eng.*, Munich, Germany, 1980, pp. 412-422.
- [6] J. B. Munson and R. T. Yeh, IEEE Software Productivity Workshop Rep., San Diego, CA, Mar. 1981.
- [7] P. F. Elzer, "Some observations concerning existing software environments," DORNIER Systems GmbH, D-7990 Friedrichshafen, Germany, Arlington, VA: Defense Advanced Research Projects Agency, May 1979.
- [8] G. Mebus, "A software engineering environment (SEE) for weapon system software—Functional description for the code and test phase," U.S. Naval Air Development Center, Warminster, PA, Rep. NADC 82183-50, Nov. 1982.
- [9] R. J. Pariseau, "A screening criterion for delivered source in military software," U.S. Naval Air Development Center, Warminster, PA, Rep. NADC-79163-50.
- [10] D. Lefkowitz, "The applicability of software development methodologies to naval embedded computer systems," Univ. Pennsylvania, Contract N62269-81-C-0455, 1982.
- [11] N. Wirth, "Lilith: A personal computer for the software engineer," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, Mar. 1981, pp. 2-15.

9



H. G. Stuebing received the B.S. degree in physics and mathematics from Ursinus College, Collegeville, PA, in 1958. After completing graduate work in physics he then received the M.S. degree from the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, where he studied computer engineering.

He joined the U.S. Naval Air Development Center and from 1958 to 1965 he worked on real-time hybrid computer simulations, particularly those concerned with astronaut training on the Navy's human centrifuge. Since 1965 he has been associated with airborne weapon system computers and software, specializing in programming languages, software engineering environments, and real-time executives. Under his direction the first integrated software engineering environment, FASP (facility for automated software production), was developed and used for weapon system software. His group has investigated the area of software reliability and developed advanced software testing methods. The group also developed a distributed real-time executive for airborne systems that has an automatic degraded mode capability.

Mr. Stuebing is a member of the ACM, and several Navy committees, including the Navy Ada Development Review Group.

## Research on Structured Programming: An Empiricist's Evaluation

IRIS VESSEY AND RON WEBER

**Abstract**—In spite of the widespread acceptance by academics and practitioners of structured programming precepts, relatively few formal empirical studies have been conducted to obtain evidence that either supports or refutes the theory. This paper reviews the empirical studies that have been undertaken and critiques them from the viewpoints of the soundness of their methodology and their ability to contribute to scientific understanding. In general, the evidence supporting programming precepts is weak. A framework for an ongoing research program is outlined.

**Index Terms**—Design, experimentation, human factors, languages, performance.

### I. INTRODUCTION

As a basis for improving the quality of software, the precepts of structured programming are compelling [22]. For the academician, the mathematics of software have a new-

found elegance and rigor. For the practitioner, structured programming concepts have strong intuitive appeal. As a result, the area has spawned a multitude of disciples, and many books, articles, and courses have appeared, all offering some dose of the new elixir.

Whereas conceptual developments in structured programming have been forthcoming, corresponding empirical developments have been slower. Several researchers have bemoaned the unsubstantiated nature of the theory [2], [24]. Since ultimately programming is an empirical science, the acid test of a normative theory of programming must be whether or not the principles derived from the theory produce cost-effective changes in software practice. In the final analysis the theory of structured programming amounts to nothing more than an interesting intellectual exercise if these cost-effective changes do not result.

In this paper we examine the precepts of structured programming from the stance of the empiricist. Our purpose is singular: we seek to show that these precepts, which many re-

manuscript received November 10, 1982; revised August 19, 1983.  
The authors are with the Department of Commerce, University of Queensland, St. Lucia, Qld. 4067, Australia.

## 10

searchers and practitioners hold fervently, are in a poor state empirically. Our intent is not to be destructive, for we count ourselves among the disciples of structured programming. Rather, we examine the nature of the underlying problems in obtaining empirical support for structured programming, suggest why these problems exist, and indicate how they might be remedied, at least in part.

The paper proceeds as follows. Section II examines structured programming theory and evaluates how well it enables empiricists to generate rigorous hypotheses about the effects of structured programming on software practice. Section III evaluates the nature of the hypotheses that have been proposed so far, especially in terms of their ability to contribute toward understanding versus prediction. Section IV surveys the empirical work that has been undertaken on structured programming and argues that the problematic results are a manifestation of poor theory, poor hypotheses, and poor methodology.

## II. STATUS OF THE THEORY

It is perhaps surprising that as empiricists we start with an examination of the status of structured programming theory. The empiricist cannot ignore the state of theory in an area. Good theory is a prerequisite to good empirical work: it increases the likelihood of any empirical research undertaken being successful; it enables the strategic propositions to be identified and to be tested; and it enables the empiricist to direct research toward understanding as well as prediction.

So far the theoretical work undertaken on structured programming has tended to follow two streams. The first stream, which we label the "characteristics" stream, is typified by the work of Dijkstra [13] and Boehm and Jacopini [6]. It seeks to show, for example, that any program can be written using certain well-defined control structures, or that a program written using these control structures can be proved correct [27]. While the characteristics theoreticians hold implied beliefs about the effects of structured programming on software practice, so far they do not seem to have formally articulated these beliefs as laws of interaction linking use of structured programming with software practice performance criteria. As such, their work is only of passing interest to the empiricist as it provides little basis for designing studies aimed at testing the effects of structured programming on practice.

The second stream, which we label the "effects" stream, does attempt to model how use of structured programming might affect software practice. Unfortunately, it has made little progress. One needs to be careful to distinguish between the rash of *claims* made for the effects of structured programming on practice, and *models* that carefully articulate relationships among the variables of interest. Indeed, from reviews of the literature and conversations with colleagues, we are still unaware of any models of this latter type; for example, a model that explains why, in a given time period, for a structured and an unstructured program containing the same bugs, a programmer is able to discover more bugs when examining the structured program. Nevertheless, the rudiments of such theories now exist. For example, Tracz [42] and Frost [17] point out how the results of psychological research on human

information processing might provide substance to the claim that structured programming facilitates programmers understanding the logic of program code—in particular, the limitations of human short-term memory seem relevant (e.g., [31], but see also [11]). And, in the area of complexity theory, Simon [41] long ago argued that systems which survive have three characteristics: they are organized as a hierarchy of subsystems, their subsystems are loosely coupled, and the internal components of a single subsystem are tightly cohesive (see also [32], [33], [48]).

There is still a substantial gap, however, between casual theorizing based on psychological concepts or complexity theory and models that attempt to define precisely the relationship between, say, structured code, chunking in short-term memory, and some software quality characteristic. Indeed, a mammoth task remains. Boehm *et al.* [5], for example, identify 11 characteristics by which software quality can be assessed. Theories must be constructed linking those psychological concepts or system complexity concepts thought to be relevant with each of these quality characteristics. Aside from the difficulties involved in fleshing out the nature of the relationships that might exist, there are onerous time requirements to investigate empirically each of the candidate relationships identified. In this sense, therefore, structured programming theory is in a sorry state, and this becomes even more apparent when we examine in the next section the status of structured programming hypotheses.

## III. STATUS OF HYPOTHESES

Given our conclusions on the empirical status of structured programming theory, it is unlikely that high-quality hypotheses will have been proposed and tested. Hypotheses are generated off a theory; the quality of hypotheses and the quality of theory are inextricably bound. Nevertheless, an analysis of the status of structured programming hypotheses provides some valuable insights into why effects theory and empirical research have been floundering.

Ultimately, the empiricist's objective is to evaluate a theory by testing hypotheses generated from the theory. In attempting to accomplish this objective *economically*, there are two concerns: first, the extent to which a test provides predictive power versus understanding; second, the extent to which a test evaluates a *strategic* hypothesis.

In terms of the first objective, ideally the goal of science is to provide understanding *and* predictive power with respect to some phenomena. As Dubin [14] points out, however, in practice these joint goals are rarely achieved concurrently. Typically, understanding is obtained without predictive power, or predictive power is attained but understanding is not enhanced. In brief, he explains the paradox as follows. We can achieve predictive power by knowing enough history about the variables of interest. Given this history, statistical methodologies can be used to predict one system state based on another system state. Precise prediction may depend only on precise description of system states and precise measurement of system states, not understanding. Understanding, on the other hand, usually can be obtained only by limiting the domain of phe-

TABLE I  
FRAMEWORK FOR ANALYSIS OF RESEARCH LEVELS OF ABSTRACTION

11

Phase	Activity	New Program Development	Repair Maintenance <sup>1</sup>	Adaptive Maintenance <sup>2</sup>	Perfective Maintenance <sup>3</sup>
	Problem Planning/Analysis				
	Program Design				
	Coding				
	Testing/Debugging				
	Documentation				
	Implementation/Delivery				

Notes: 1. Correction of logic errors in released programs.  
2. Alterations carried out to meet changed program specifications.  
3. Alterations to improve resource consumption efficiency.

nomena analyzed, deliberately simplifying the domain, and focusing on broad relationships. These strategies limit the predictive power of the model devised, however.

Blalock [3] characterizes the problem in terms of choosing the appropriate "level of abstraction" during theory construction. If the level of abstraction is too high, it is difficult to make the theory operational in the sense of developing testable propositions comprising variables that can be defined and measured precisely. As more variables enter a theoretical model, complexity of interactions increases, and executing a test of the model is no longer straightforward. Nevertheless, too much reductionism must be resisted in that it leads to trite theoretical models. Care must be taken, however, not to generalize and to abstract to a level that is untestable. Hence, researchers strive for "middle range" theories—theories that are neither too general nor too specific.

In terms of the empiricist's second objective during hypothesis testing, the search for strategic hypotheses is motivated by a desire for parsimony—a desire to prove, improve, or disprove a theory quickly with minimum effort. Strategic hypotheses are those hypotheses that deal with "notable happenings" in the values of the variables of interest. For example, if the interaction between two variables is represented by some polynomial function, strategic hypotheses would deal with maxima, minima, and points of inflection. Recall that in the previous section we emphasized that a large number of possible relationships may need to be enunciated in a theory of structured programming. Hence, many propositions to be tested may be generated off the theory. Furthermore, each proposition may need to be investigated in terms of several metrics existing for the variables of interest. For example, Boehm *et al.* [5] identify 151 metrics for their 11 quality characteristics. Clearly, testing the strategic hypotheses first is critical if parsimony is to be achieved.

It is our contention that, with few exceptions, the status of the "effects" theories in structured programming is such that they do not contribute to understanding nor do they facilitate selecting strategic hypotheses. In general, the theories have

sought predictive power, though they have not been especially successful in this respect, either.<sup>1</sup>

As a basis for our arguments, consider, first, Table I. The rows represent major phases within the programming process; the columns represent the various types of programming activities [1]. In a more extensive analysis, further dimensions to the matrix would be added; for example, dimensions to represent program complexity or the way in which programming teams are organized.

Consider, now, two levels of abstraction. At a moderately high level of abstraction, assume we are trying to develop theory and generate hypotheses relating use of structured programming and various quality attributes of *new* programs. For example, we might employ the quality attributes and their associated metrics described by Boehm *et al.* [5]. Thus, we would attempt to describe the interactions between structured programming and portability, structured programming and reliability, structured programming and efficiency, etc., and then generate hypotheses based on our model. Note we are dealing with a *column* of the matrix.

Alternatively, we might attempt to build theoretical models for a phase within an activity—a *cell* of the matrix. At this lower level of abstraction, again we would define various criterion variables of interest and attempt to describe the relationships between structured programming and these variables. For example, for the coding phase in new program development, we might attempt to model the effects of structured programming on the understandability of the code.

While research at the column level may contribute to our

<sup>1</sup>As an aside, it would be interesting to examine the plethora of claims made for structured programming and analyze them from a prediction/understanding perspective. We have not attempted an exhaustive investigation of all the literature on structured programming, and so our views are colored by that particular subset of the literature with which we are familiar. Nevertheless, it seems to us that the claims made are predominantly prediction-oriented rather than understanding-oriented. Perhaps this reflects that consultants/practitioners who are primarily interested in results dominate the literature rather than scientists who hopefully are more interested in understanding.

predictive powers, we argue that, at this time, research at the cell level must be undertaken if we are to improve our understanding of the effects of structured programming and our ability to identify the strategic hypotheses.<sup>2</sup> We cannot prove our point so we attempt first to marshal support for our view by example. Suppose several replications of an experiment show the following result: if structured programming is used to produce a new program, the total number of labor hours expended will be less than the number expended to produce a program of equivalent quality when structured programming is not used. Scientists are now compelled to ask: why do these results hold true? To answer this question, they must induce the laws of interaction between structured programming and labor hours expended to produce a new program. It quickly becomes apparent that some complex interactions exist. A researcher might be confident that decreased testing time is associated with use of structured programming, but they may *not* be confident that decreased coding time is associated with use of structured programming. If the possibility exists that structured programming is associated with both increases and decreases in labor hours expended on the various programming phases, will it always be true that use of structured programming is associated with decreased development time? For "simple" programs, for example, is it possible that structured programming is associated with increased labor hours expended? Indeed, Kernighan and Mashey [23] question the usefulness of structured design techniques for simple programs.

In attempting to answer these questions, researchers are forced into a process of disaggregation. Whenever the relationship between two variables must be explained, it can only be accomplished in terms of a process description incorporating variables on a lower level of abstraction. In theory construction, when, therefore, is this disaggregation process likely to stop? We argue it stops when researchers are confident in the validity of the assertions underlying the theory proper and the derived relationships of interest. These assertions constitute the axioms of the theory.<sup>3</sup>

To illustrate the process, consider again the previous example. In attempting to explain why decreased labor hours expended on new program development were associated with use of structured programming, researchers might give the following propositions.

- 1) Two phases that comprise the new program development process are coding and testing (axiom).
- 2) Decreased labor hours expended on coding are associated with use of structured programming (axiom).
- 3) Decreased labor hours expended on testing are associated with use of structured programming (axiom).
- 4) Decreased labor hours expended on new program development are associated with use of structured programming (derivation).

<sup>2</sup>We recognize that certain theories are pathbreaking in terms of their generality rather than their specificity; for example, Einstein's theory of relativity. However, these theories tend to be based on a set of restricted theories and empirical works that are already well advanced.

<sup>3</sup>Once the axioms of a theory are stated, the derivations (theories) are a logical consequence. If the derivations prove wrong, assuming the system of logic has been applied correctly, then the axioms must be wrong.

Clearly this theory stands or falls on the basis of the truth of its axioms.<sup>4</sup> If researchers are confident that the axioms are true, the disaggregation process stops. If there are doubts about one or more of the axioms, the disaggregation process continues, the axiom becomes a derivation, and new axioms are formulated in terms of variables expressed at a lower level of abstraction.

Unfortunately, the claims (propositions) made for structured programming are not logical consequences of a carefully constructed theory. We are not aware of any set of axioms and derivations; consequently, it is impossible to evaluate the axioms on which the claims rest. Moreover, it is apparent that the claims tend to be made at a fairly high level of abstraction. Even if it is possible to induce the theoretical model used as the basis for the claims, in light of our arguments above, we conclude that the axioms will be disputed.

To further illustrate some of these problems, consider the claim made by Yourdon [47] that structured programming will lead to fewer testing problems. From an empirical research perspective, this claim is hopelessly vague. Yourdon gives some indication of the criterion variables he has in mind: he believes structured programming will reduce the effort and cost of testing large programs and enable large systems to be released with fewer bugs.

Consider, first, the issue of effort and cost. We do not know what is meant by effort, but presumably it covaries with cost, so we examine cost as the criterion variable. Yourdon's focus is *large programs*. However, it is unclear whether he is concerned only with the testing phase for new programs (a single cell in Table I) or with the testing phase for any of the four types of programming activity (a row—four cells in Table I). If his focus is a single cell, although he does not say so, it seems possible to induce a theoretical model based on relationships among cost of testing, ease of program understanding, understanding and memory clustering, clustering and structured programming, etc. If his focus is a row of Table I (a higher level of abstraction), more uncertainty is likely to exist about the validity of any relationship posited in the theory. For example, are the processes required to understand a program the same for new program development and perfective maintenance? If not, does structured programming facilitate both types of understanding required?

Next, consider the issue of the number of bugs existing in large program releases. The criterion variable is a quality characteristic of the final product of the programming process rather than the product of each phase. Again, it is unclear whether we are dealing with only new program releases (a single column of Table I) or all types of programming activities (four columns of Table I). Given the proposition, it is an insightful but frustrating exercise to try to induce the underlying theoretical model. Nevertheless, assuming that empirical evidence shows the proposition to hold (it has predictive power), from an understanding perspective, why does it hold? We are certain that the response of researchers attempting to

<sup>4</sup>Blalock [3] points out that an axiom in the empirical sciences is different from an axiom in mathematics. In mathematics an axiom is a truth statement taken for granted. In the empirical sciences it is an assumption that is "almost universally accepted" (our emphasis).



answer this question is that they will commence the disaggregation process and analyze the effects of structured programming or the introduction of bugs during each phase of the programming process.

We do not mean to be disparaging in our analysis of Yourdon's claim. The development of theory in an area typically requires the concerted efforts of many researchers. What we have attempted to do is convince, through example, that the claims made for structured programming are in a sorry state, primarily because the underlying theory on which the hypotheses depend is weak or nonexistent and the choice made of a level of abstraction for the research so far has been inappropriate if understanding is to be obtained. Insofar as the propositions and the underlying theory remain in a primitive state, it is unlikely that empiricists can design experiments or undertake surveys or field studies that contribute meaningfully to our understanding the effects of structured programming on software practice. Furthermore, it is unlikely that the empirical research can proceed in a parsimonious fashion since the strategic hypotheses cannot be identified.

#### IV. EMPIRICAL STUDIES

In this final section of the paper, we survey the empirical studies undertaken, with which we are familiar, on the effects of structured programming on software practice. Our conclusions on the status of structured programming theory and hypotheses do not auger well for the success of empirical studies. Nevertheless, the empiricist is concerned with the empirical studies carried out in an area for three reasons. First, the results indicate whether the theory seems to have potential in terms of explaining or predicting phenomena and is, therefore, a fruitful one to pursue. Second, there is a concern with whether the results obtained have contributed to both understanding and predictive power. Third, especially if the results of different studies conflict or the results conflict with the theory, there is a concern with the quality of the empirical research methodology used.

The following three subsections examine briefly the laboratory studies, field studies, and surveys undertaken on the effects of using structured programming on software practice. Most of the analysis is couched in terms of the framework we established in the previous two sections on theory and hypotheses; namely, whether a theory underlies the hypotheses tested and whether the level of abstraction chosen facilitates understanding or prediction. We have treated traditional methodological issues in a cursory way, in spite of their importance. In the fourth subsection that follows, these issues are addressed by way of a summary critique and evaluation.

##### A. Laboratory Studies

Table II provides an overview of the laboratory studies undertaken on the effects of using structured programming on software practice. Even a cursory examination of the table shows the studies completed so far have been limited in terms of the tasks examined, the types of subjects used, the types of programs used, the programming languages investigated, etc. [38]. Furthermore, the summary results column shows that the support obtained for structured programming is problematic.

The series of studies by Sime, Green, and Guest examined the effects of four control structures—a branch-to-label (GO-TO) and three versions of a nested conditional (IF-THEN-ELSE)—on various performance measures. In general, the results support use of the nested conditional over the branch-to-label, although one version of the nested conditional caused syntax problems and inhibited subjects producing an error-free program on the first attempt. The programming languages used were artificial languages developed specifically for the experiment. Their grammars were very limited.

Sime, Green, and Guest had their subjects undertake program composition (design, coding, and testing) and program comprehension tasks. In terms of the level of abstraction issue discussed earlier in the paper, their research provides an interesting case. To illustrate, consider their 1977 study [40]. The composition task involved three phases: design, coding, and testing. The program specifications were given, and documentation and implementation were not required. The independent variables manipulated were control structure and indentation—the branch-to-label program had no indentation. At first glance, therefore, according to our previous arguments, the research should provide predictive power rather than understanding because it was not confined to a cell in Table I.

Note, however, the dependent variables used. Some are joint performance measures for two or three programming phases while others are performance measures for one phase only (see Table I). The number of semantic errors and the number of error-free programs are functions of how well the design and coding phases were performed. Composition time is a function of the time consumed in the design, coding, and testing phases. The number of syntax errors, however, is a function of how well the coding phase was performed, and error lifetimes may be a function of how well the testing phase was performed; that is, for both criterion measures, they apply to one programming phase only (a cell in Table I).

Consider, first, the results obtained for error lifetimes. One version of the nested conditional outperformed the branch-to-label and the other version of the nested conditional. If we are willing to make the (heroic) assumption that the frequency and distribution of error types is the same across languages after the first compile, error lifetimes is a performance measure for the testing phase only. Sime, Green, and Guest asked the inevitable question: what aspect of the control structure facilitated testing? They induced that there were two types of tasks involved in programming: classifying information taxonomically and converting taxa into a linear sequence. The first task involves identifying the actions to be performed given a set of conditions are fulfilled. The second task typifies the coding phase: converting the taxonomy into program instructions. During testing, both tasks must be performed. Sime, Green, and Guest argued the differences in error lifetimes reflected that the control structures facilitated differentially the extraction of taxon information. This was an important leap in *understanding*. The level of abstraction was right! Only a cell in Table I had been investigated.

In contrast, consider the result for the number of semantic errors—the number of semantic errors was less for the nested conditionals. In our view it is impossible to determine whether

TABLE II  
LABORATORY STUDIES IN STRUCTURED PROGRAMMING

Author(s)	Task	Subjects	Language(s)	Program Size (LOC)	Independent Variable(s)	Dependent Variable(s)	Results for Structured Programming
Sime, Green, and Guest [39]	Program composition	Naive subjects (18)	JUMP NEST	10	Control Structure Indentation	Semantic errors Syntactic errors Error-lifetimes Time No. subjects completing	+ 0 0 + +
Weissman [46]	Program understanding	Graduate students (16)	PL/I	46-85	Control flow Indentation	First self-evaluation Quiz Accuracy of hand simulation Second self-evaluation Quiz repeated Time	0 0 0 0 0 0
Weissman [46]	Program understanding	Students (24)	PL/I	44-46	Control Flow Indentation	First self-evaluation First quiz Second self-evaluation Accuracy of Modifications Third self-evaluation Second quiz	+ 0 + 0 + 0
Lucas and Kaplan [29]	Program composition	Students (32)	PLC		Control structure	No. runs Time Compile time Object code size Ease in composition Ease in debugging Enjoyment	- 0 0 0 - 0 0
Lucas and Kaplan [29]	Program modification	Students (32)	PLC		Control structure	No. runs Time Compile time Object code size Ease in composition Ease in debugging Enjoyment	0 + + + + 0 0
Sime, Green and Guest [40]	Program composition	Naive subjects (45)	JUMP NEST-BE NEST-INE	10-25	Control structure Indentation	Semantic errors Syntactic errors Error-free programs No. of runs Time	+ - - + 0
Green [19]	Program comprehension	Experienced programmers (12)	JUMP NEST-BE NEST-INE		Control structure Indentation	Time	+
Love [28]	Program understanding	Inexperienced students (10)	FORTRAN	13-20	Control flow Indentation	% LOC recalled	0
Love [28]	Program understanding	Experienced students (12)	FORTRAN	13-20	Control flow Indentation	% LOC recalled Rated understanding	+ 0
Sheppard et al. [36]	Program comprehension	Experienced programmers (36)	FORTRAN IV	26-57	Design Control flow	No. statements correctly recalled	+
Sheppard et al. [36]	Program modification	Experienced programmers (36)	FORTRAN IV	39-56	Design Control flow	Accuracy Time	+ 0
Sheppard et al. [36]	Program debugging	Experienced programmers (54)	FORTRAN IV	25-225	Design Control flow	Time	0

this can be attributed to better program design, better program coding, the existence of indentation, the nature of the control structure used, or the existence of an interaction effect; that is, multiple cells in Table I are confounded. Program design involves extracting taxon information. Coding involves converting taxon information into sequence information. How do the design and coding processes proceed? One alternative is that the programmer designs first and then codes. Another alternative is that design and coding proceed concurrently. If this second alternative is the case, the different control structures may facilitate design rather than coding—hence, the differences in the number of semantic errors that resulted. If programmers started with a common design (taxa), perhaps there would be few differences in the number of semantic errors resulting using the three languages. While prediction has been enhanced, understanding has not.

Weissman [46] studied the effects of control structure and indentation (paragraphing) on program understanding. In his first experiment he found no effects on several criterion variables that he used (see Table II). In his second experiment he found a positive effect for control structure on three self-evaluations of understanding and a positive effect for indentation on the first self-evaluation.

While his research provides some support for structured programming, it contributes little to understanding why structured programming may facilitate program comprehension.<sup>5</sup> Clearly, program comprehension is an important prerequisite to performing different types of programming activities. But different types of comprehension may be required to perform, say, repair maintenance versus adaptive maintenance. For example,

<sup>5</sup>Weissman's experiments also have some important methodological problems. See, e.g., [37].

to correct a bug, only localized comprehension may be required, whereas to modify a program to better meet user needs, global comprehension may be required. Thus, comprehension needs to be studied at the cell level in Table I rather than the column level.

Lucas and Kaplan [29] investigated program composition and program modification. They used an experimental group and a control group, the subjects being students with some programming experience. The only difference between the two groups was that the experimental group could not use the GO-TO. Results for the composition task did not support abolition of the GO-TO.<sup>6</sup> Results for the modification task, however, did support abolition of the GO-TO.

From their dependent variables, it is difficult to induce the theoretical models that motivated their choices. What, for example, is the nature of the relationship between abolition of the GO-TO and compile time or object code size? Moreover, in terms of the level of abstraction used, it is difficult to see how their research would contribute to understanding. Again, they worked at the column rather than the cell level of Table I.

Love [28] examined the effects of control flow (including structured constructs) and indentation on a subject's ability to understand a program, measured by the percentage of lines of code correctly recalled after the program was studied. For students in an introductory Fortran class, he found neither structured constructs nor indentation had an effect. For graduate computer science students, he found that use of structured constructs had a positive effect. Indentation, however, still had no effect. In addition, in light of results with the undergraduate students, he asked the graduate students to write a one or two sentence explanation of the function of each program module. These explanations were scored on a 5-point scale as a measure of understanding. Neither control flow nor indentation had an effect. As an aside, Love found a high correlation (0.67) between the subject's ability to recall a statement and the logarithmic transformation of the size of the largest program they had written.

As a program quality measure, where program understanding can be placed in terms of Table I is somewhat uncertain. Consider the three types of maintenance activities shown. From a columnar perspective, presumably understanding is a prerequisite to effective analysis, design, and testing when maintenance must be carried out. Does structured programming have a positive effect for each of these phases? From a row perspective, are different types of understanding required for the different types of maintenance activities? Again, prediction versus understanding is the research design issue at hand.

The three studies by Sheppard *et al.* [36] manipulated control flow (including structured constructs) in three tasks—comprehension, modification, and debugging. They found support for structured programming in terms of the number of statements correctly recalled and the accuracy of modifications, but no support in terms of the time taken for modifications and the time taken for debugging.

<sup>6</sup>These results are suspect. The students received no "special" training in structured programming, and Lucas and Kaplan found evidence of a learning effect.

Their results emphasize the need to disaggregate the attributes of a structured program—control flow, control structures, indentation—if understanding is to be obtained. Sheppard *et al.* used three versions of a program: one with a convoluted structure, one with a "naturally structured" control flow, and one with a "strictly structured" control flow. In the comprehension task the naturally structured and strictly structured control flows outperformed the convoluted version, but there was no difference in performance between the naturally structured and strictly structured versions. Similar results were obtained for the modification task.

From an understanding perspective, the problem that now exists is to determine the extent to which control flow (program design) versus control structure affects the results obtained.<sup>7</sup> Since no difference existed between the naturally structured and strictly structured versions, if the control flow (designs) were the same or similar, the variation in control structure seems to have had little effect. But it seems the convoluted program represents a variation in *both* the control flow and the control structures used; thus the separate effects cannot be disaggregated.

The levels of aggregation problem also exists in terms of the dependent variables. Consider, for example, the dependent variable "accuracy" in the program modification task. Given that the programmers were able to modify the naturally and strictly structured programs more accurately than the convoluted program, was this the result of them being able to design more accurate modifications, code more accurate modifications, or both?

In summary, the results of the laboratory studies on structured programming are equivocal. Moreover, in terms of future research, many of the cells in Table I have yet to be investigated, only certain software quality attributes have so far been examined, and, in our opinion, because of the levels of abstraction problem, the existing research contributes more to prediction rather than to understanding.

### B. Field Studies

Table III provides an overview of the field studies undertaken on the effects of using structured programming on software practice. Our analysis of field studies will be brief since, in general, the nature of field studies prohibits them from contributing much to understanding as opposed to prediction. Moreover, little theory underlies the research undertaken so far.

Of the four studies listed in Table III, only two provide evidence in favor of structured programming. Walston and Felix [45] collected data on 60 completed software development projects and examined 68 variables to determine whether they correlated significantly with productivity measured as the ratio of delivered lines of source code to total effort in worker-months. Twenty-nine variables showed a significantly high correlation including use of structured programming and

<sup>7</sup>This problem also underlies the work by Weissman [46] and Love [28]. Weissman, for example, attempted to manipulate the complexity of control flow independently of using structured constructs. He found he could not develop a program using structured constructs that had a complex control flow. McCabe [30] provides a formal analysis showing the difficulty (impossibility) of manipulating control flow independently of using structured constructs.

TABLE III  
FIELD STUDIES ON STRUCTURED PROGRAMMING

16

Author(s)	Dependent Variable	Program Characteristics	Results for Structured Programming
Walston and Felix [45]	Productivity	28 high-level languages 66 computers 4000-467,000 LOC 12-11527 worker-months.	+
Vessey and Weber [44]	Logic Errors	Commercial COBOL programs	+
Lawrence [25]	Productivity	Commercial COBOL programs	0
Vessey [43]	Development Time Productivity	Commercial COBOL programs	0

TABLE IV  
SURVEYS ON STRUCTURED PROGRAMMING

Author	Survey Characteristics	Claimed Advantages of Structured Programming
Holton [20]	33 large organizations in Los Angeles area	More efficient debugging and testing Better quality programs Easier and less costly maintenance Clearer and more useful documentation Lower development costs
Hugo [21]	309 organizations worldwide	Reduced debugging time and computer time for testing Reduced project elapsed time Reduced project labor time Reduced errors made Reduced maintenance
Lientz and Swanson [26]	487 data processing managers	Better design specifications Improved quality of original programming Improved documentation

use of top-down development.<sup>8</sup> Both produced a positive effect on productivity.

Vessey and Weber [44] collected data on 447 operational commercial and clerical Cobol programs in an Australian organization and two U.S. organizations. They examined whether use of structured programming affected the extent of repair maintenance needed on a program; that is, maintenance needed to correct logic errors identified after the program had been released initially or maintained subsequently. In the Australian organization they found top-down (modular) design resulted in a decreased amount of repair maintenance. Because of insufficient data, however, the effects of the three structured control constructs could not be investigated. For the U.S. organizations, neither top-down design nor the control structures had any effect. Perhaps the most important finding of their study was that repair maintenance activities were infrequent for the three organizations studied. For the Australian organization, 90 percent of programs had two or fewer repairs carried out; for the U.S. organizations, 92.1 and 94 percent were the corresponding figures. This does not auger well for a claim to the effect that a significant practical advantage of using structured programming is a decrease in a number of errors existing in released programs.

Lawrence [25] collected data on 278 commercial programs from 23 medium- to large-scale Australian organizations. He examined the effect of using structured programming on productivity, defined as the ratio of the number of procedural lines of source code to worker-hours. He found no evidence in support of top-down design or structured control constructs as a means of increasing programmer productivity.

Vessey [43] studied 353 Cobol programs from three commercial organizations to determine whether use of structured programming (top-down design plus structured control constructs) affected the time to develop programs and programmer productivity defined as the ratio of the number of procedure division lines of code to worker-hours. She found no evidence to support use of structured programming.

In summary, the empiricist would not be encouraged by the support found for use of structured programming in the field studies carried out. Is further empirical research worth pur-

suing? Have methodological problems in the existing studies nullified the results? Have the "right" questions been asked? Again, we argue that the answers to all three questions are bound up in the need for a more extensive "effects" theory of structured programming to be enunciated. Without this theory the empiricist tests obscure hypotheses, is unclear about the variables of interest, is uncertain how variables should be defined, measured, and controlled, and has little sense of the strategic hypotheses to test. The existing studies do little to contribute to understanding and an "effects" theory. They have all been undertaken at a fairly high level of aggregation, aimed more at prediction rather than understanding.

### C. Surveys

Like field studies, surveys, by their very nature, are unlikely to contribute much to understanding. Indeed, there are fewer chances to enhance understanding in that opportunities for follow-up are more constrained than for field studies. Unless questions on the survey materials have been designed specifically to enhance understanding or respondents have, and avail themselves of, opportunities to explain why an effect occurs, surveys give only a global results picture.

Table IV provides an overview of the surveys undertaken on structured programming. Holton [20] surveyed 33 large organizations in the Los Angeles area and found 23 of them had implemented "improved programming technologies" sufficiently for them to be able to report their experience. Using a 4-point scale, he asked users to rate the impact of the various technologies on software, programming staff, and users. He found that respondents claimed use of structured programming (defined in terms of the three control structures) primarily resulted in more efficient debugging and testing, better quality (e.g., more error-free) programs being produced, and easier and less costly program maintenance. Use of top-down design and implementation primarily resulted in more efficient debugging and testing, clearer and more useful programming system documentation, and lower development costs.

Hugo [21] reports a worldwide survey of 309 organizations using the so-called improved programming technologies. In terms of structured code, the primary benefits claimed were

<sup>8</sup>From a statistical viewpoint, their study suffers from problems of multicollinearity.

reduced debugging time and computer time for testing, reduced project elapsed time, reduced project labor time, reduced errors made, and reduced maintenance. In terms of structured design, the primary benefits claimed were a decrease in the number of errors made and a decrease in the labor time spent on debugging.

Lientz and Swanson [26] surveyed 2000 data processing managers. From 487 responses received, they found that users claimed structured programming resulted in more adequate design specifications, improved quality of the original programming, and improved documentation quality. Nevertheless, they found no relationship between use of structured programming, user knowledge, and programmer effectiveness. This finding was especially important since, in a factor analysis of 26 program maintenance problem items, user knowledge and programmer effectiveness accounted for 71.4 percent of the common problem variance.

Subject to the normal qualifications about comparing surveys comprising different respondent populations, using different response scales, etc., there are some common findings in the surveys by Holton and Hugo: users see use of structured programming resulting in programs being produced with fewer errors, decreased debugging and testing costs, and decreased maintenance costs. However, their studies give little insight into why these results occur. Conversely, Lientz and Swanson chose their problem items at a level of aggregation that gives some insight into why the benefits found by Holton and Hugo might occur: better design, better programming (coding?), better documentation. Of course, why structured programming produces better design, better coding, and better documentation is another issue, but again, we emphasize the importance of choosing the "right" level of research aggregation if understanding is to be obtained.

As a final point, the research by Lientz and Swanson raises another issue that is an overriding concern for structured programming research; namely, that user knowledge and programmer effectiveness seem to have important effects on software maintenance costs. This finding is supported by other research. For example, Boehm *et al.* [4], Rubey *et al.* [35], and Glass [18] found a large proportion of errors in several software projects they examined to be design errors that arose because the software did not comply with user requirements (see also [10]). Structured programming can do little to remedy this problem. Structured *analysis*, however, *does* attempt to improve communications between the designer and user [12]. In this light, therefore, research on structured analysis may have greater practical importance.

#### D. Overall Critique and Evaluation

We return to the three questions asked at the beginning of this section. Is there sufficient evidence to show that further empirical research is worth pursuing? Have the results obtained contributed to understanding versus prediction? Is the empirical research methodologically sound?

In terms of the first question, in our opinion the empirical results are supportive of structured programming but not compelling. There are few negative results; they show either no effect or a positive effect when structured programming is used. What seems clear, however, is that use of structured pro-

gramming does not have a strong, pervasive effect that washes out the influence of other factors on programming practice. If the effects of using structured programming are to be fully understood, careful controls will have to be exercised over any empirical research undertaken so that only a few factors are allowed to vary, at least in the initial stages of the research. Tightly controlled experiments raise another issue, however. As more and more factors are controlled in an experimental setting, the researcher is then bound to ask about the practical significance of the factors that are experimentally manipulated. Ultimately, some type of effect can be produced if enough factors are controlled! Sheil [37] points out, however, that the relative importance of those factors that are controlled versus those that are varied is an issue of real-world significance.

In terms of the second question, we have attempted to demonstrate that relatively little of the research has been designed in such a way that it contributes to understanding the effects of structured programming. Admittedly the empiricist has a poor theoretical base on which to work, but to some extent understanding still can be obtained by choosing the appropriate research level of abstraction and implementing careful experimental controls. For the most part the research has failed on the first count and, as we discuss below, it has failed on the second count also.

In terms of the third question, the empirical work that has been carried out often has been badly flawed. Sheil [37] and Brooks [9] provide excellent analyses of some major methodological problems with the laboratory studies we have surveyed. We do not reiterate these problems. Nevertheless, there is one matter mooted that we believe is central to any further empirical research undertaken; namely, whether the complexity of programmer skill precludes successful empirical research using a classical experimental approach.

A common feature of the results obtained for the empirical studies undertaken so far on structured programming is the large "within-subjects" variance and the low percentage of variation in the dependent variables accounted for by the independent variables manipulated [37].<sup>9</sup> In classical experimental terms, this means that the researcher needs to control other factors that are affecting the dependent variables if the effects of the independent variables of interest are to be better understood. Of course, the obvious question is: what are these variables that need to be controlled? Like Sheil [37] and Brooks [8], we contend that subjects often use substantially different approaches to solving programming problems, such that manipulating, say, the syntax of a programming language is unlikely to account for much of the variation in performance. Thus, it is these differences in approach that need to be controlled.

Programming is a process of applying knowledge structures (domains) to a problem to obtain a solution ultimately expressed in a programming language [7]. By a knowledge structure we mean, very loosely, a general solution method. Floyd

<sup>9</sup>This result is not common to all studies—Lawrence [25] reports comparatively small within-subjects variation. He comments that the nature of the task may be an important consideration: complex algebraic and maze problems may be quite different from commercial Cobol work.

[16] terms a knowledge structure a "paradigm" of programming. For example, two paradigms (knowledge structures) in programming are the branch-and-bound technique and the divide-and-conquer technique. An expert programmer who knows these techniques recognizes specific problems as being particular examples that are amenable to solution using the techniques. The novice programmer may confront the problem and be left floundering. Newell and Simon [34] distinguish between experts and novices precisely in this way: the elaborateness of the knowledge structures possessed by the problem solver. When a high level of skill is needed to solve a problem, these differences in knowledge structures are likely to account for a large proportion of the variance in performance among subjects. Sheil [37] argues cogently that programming is a highly skilled activity. He contends [37, p. 118]: "However programmers' knowledge bases are actually organized, their existence and size seem clear. Hypotheses which posit differences in either individual aptitude or task difficulty are therefore, at least, extremely difficult to investigate, as the enormous size of the knowledge bases being drawn on imply (sic) that different individuals approach the same task with vastly different resources."

Consider, again, the nature of the phases involved in the programming process. Clearly, at least some of the phases require high-level skills that traditionally have been acquired through experience rather than taught formally; for example, the design and testing/debugging phases, and perhaps, at times, even the coding phase [7]. If, for example, researchers attempt to investigate whether a change in some programming language feature has a beneficial effect on programming performance, they must ensure that subjects in their experiments use the same knowledge structure(s) to solve the problem. Otherwise, it is impossible to know whether performance variations can be attributed to the changed language feature. The "real" question the researcher is attempting to answer is: given a programming paradigm, how well does the language feature implement the paradigm? Floyd [16, p. 458] expresses this point more eloquently: "I believe that the continued advance of programming as a craft requires development and dissemination of languages which support the major paradigms of their user's communities. The design of a language should be preceded by enumeration of these paradigms, including a study of the deficiencies in programming caused by discouragement of unsupported paradigms." Unless the paradigm is controlled in any empirical research, therefore, the language features will defy systematic study.

How, then, can classical experimentalists determine whether their subject groups have homogeneous knowledge structures when they attempt a particular programming problem? One methodology that might be used is protocol analysis—having subjects talk aloud when undertaking a programming task and then analyzing their protocols to identify similarities and divergences [7]. The methodology is still new, and it has its share of problems [15]; nevertheless, in terms of a difficult task, it is an important start.

In light of these methodological difficulties, consider, finally, the nature of structured programming. Is it an attempt to modify knowledge structures? Or an attempt to improve pro-

gramming languages that implement certain knowledge structures? Floyd [16] clearly sees structured programming as manipulating knowledge structures; it is the "dominant paradigm in most current treatments of programming methodology (our emphasis). In our view the answer is not clear-cut; some extent it depends on how one defines structured programming. While we agree that top-down design and adherence to the three structured control structures constitute a very general paradigm for solving a problem, a characterization of structured programming to include rules of indentation, commenting, etc., may mean that implementation issues rather than knowledge structure change issues are being considered. In any case, whenever researchers are manipulating a variable considered to be an attribute of structured programming, it behooves them to ask whether they are manipulating knowledge structures or language implementation modes or both. If researchers are manipulating knowledge structures, subjects must be adequately trained in the knowledge structure. If the researcher is manipulating a language implementation mode, they must ensure that the subjects used have homogeneous knowledge structures if the results are not to be confounded.

## V. SUMMARY AND CONCLUSIONS

In this paper we have surveyed and analyzed the existing research on structured programming from the stance of an empiricist. We have proffered four major arguments: first, that the theory enunciating the effects of structured programming on software practice is rudimentary and inadequate; second, that this lack of theory has inhibited the formulation of hypotheses that contribute to both understanding and predictive power; third, that until the theory has been developed, it is not possible to identify the strategic hypotheses and, as a consequence, carry out parsimonious empirical research; and finally, that the existing empirical work reflects the shoddy state of the theory in that it does not effect a coordinated whole, nor has it aspired to understanding as opposed to prediction.

The precepts of structured programming are compelling, yet the empirical evidence obtained so far is equivocal. Good empirical research is inextricably bound to the existence of good theory. Unfortunately, we do not seem to have understood the urgency of this relationship in terms of our current work on structured programming.

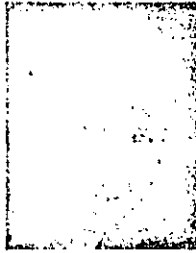
## ACKNOWLEDGMENT

The authors are indebted to A. Borning, K. Mathieson, participants in workshops at Indiana and New York Universities, and the reviewers for comments on earlier versions of this paper.

## REFERENCES

- [1] J. D. Aron, *The Program Development Process Part I - The Individual Programmer*. Reading, MA: Addison-Wesley, 1974.
- [2] V. R. Basili and R. W. Reiter, "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 299-320, May 1981.
- [3] H. M. Blalock, *Theory Construction: From Verbal to Mathematical Formulations*. Englewood Cliffs, NJ: Prentice-Hall, 1969.
- [4] B. W. Boehm, R. W. McClean, and D. B. Urfrig, "Some experience with automated aids to the design of large-scale reliable software," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 125-135, Mar. 1975.

- [5] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merit, *Characteristics of Software Quality*. Amsterdam, The Netherlands: North-Holland, 1978.
- [6] C. Bohm and G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formulation rules," *Commun. Ass. Comput. Mach.*, vol. 9, pp. 366-371, May 1966.
- [7] R. E. Brooks, "Towards a theory of the cognitive processes in computer programming," *Int. J. Man-Mach. Studies*, vol. 9, pp. 737-751, 1977.
- [8] ---, "Using a behavioral theory of program comprehension in software engineering," in *Proc. 3rd IEEE Int. Conf. Software Eng.*, New York, 1978, pp. 196-201.
- [9] ---, "Studying programmer behavior experimentally: The problems of proper methodology," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 207-213, Apr. 1980.
- [10] R. G. Canning, "Getting the requirements right," *EDP Analyzer*, vol. 15, pp. 1-14, July 1977.
- [11] B. Curtis, "In search of software complexity," in *Proc. IEEE Workshop Quant. Software Models for Reliability, Complexity, and Cost*, New York, 1980.
- [12] R. DeMarco, *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
- [13] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.
- [14] R. Dubin, *Theory Building*. New York: Free Press, 1969.
- [15] H. J. Einhorn, D. N. Kleinmuntz, and B. Kleinmuntz, "Linear regression and process tracing models of judgment," *Psychol. Rev.*, vol. 86, pp. 465-485, 1979.
- [16] R. W. Floyd, "The paradigms of programming," *Commun. Ass. Comput. Mech.*, vol. 22, pp. 455-460, Aug. 1979.
- [17] D. Frost, "Psychology and program design," *Datamation*, vol. 21, pp. 137-138, May 1975.
- [18] R. L. Glass, "Persistent software errors," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 162-168, Mar. 1981.
- [19] T. R. G. Green, "Conditional program statements and their comprehensibility to professional programmers," *J. Occup. Psychol.*, vol. 50, pp. 93-109, 1977.
- [20] J. B. Holton, "Are the new programming techniques being used?," *Datamation*, vol. 23, pp. 97-103, July 1977.
- [21] I. S. J. Hugo, "A survey of structured programming practice," in *AFIPS Conf. Proc.*, New York, 1977, pp. 741-752.
- [22] R. W. Jensen, "Structured programming," *IEEE Computer*, vol. 14, pp. 31-48, Mar. 1981.
- [23] B. W. Kernighan and J. R. Mashey, "The UNIX programming environment," *Software-Practice and Experience*, vol. 9, pp. 1-15, 1979.
- [24] R. Kling and W. Scacchi, "Computing as social action: The social dynamics of computing in complex organizations," in *Advances in Computing*, M. Yovits, Ed. vol. 19. New York: Academic, 1980.
- [25] M. J. Lawrence, "An empirical study of commercial programming productivity," Univ. New South Wales, unpublished, 1980.
- [26] B. P. Lientz and F. B. Swanson, "Problems in application software maintenance," *Commun. Ass. Comput. Mach.*, vol. 24, pp. 763-769, Nov. 1981.
- [27] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979.
- [28] L. T. Love, "Relating individual differences in computer programming performance to human information processing abilities," Ph.D. dissertation, Univ. Washington, unpublished, 1977.
- [29] H. C. Lucas and R. B. Kaplan, "A structured programming experiment," *Comput. J.*, vol. 19, pp. 136-138, 1976.
- [30] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308-320, Dec. 1976.
- [31] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychol. Rev.*, vol. 63, pp. 81-97, 1956.
- [32] J. G. Miller, *Living Systems*. New York: McGraw-Hill, 1978.
- [33] G. J. Myers, *Reliable Software Through Composite Design*. New York: Petrocelli/Charter, 1975.
- [34] A. Newell and H. A. Simon, *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.
- [35] R. J. Rubey, J. A. Dana, and P. W. Biche, "Quantitative aspects of software validation," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 150-155, June 1975.
- [36] S. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern coding practices and programmer performance," *IEEE Computer*, vol. 12, pp. 41-49, 1979.
- [37] B. A. Shell, "The psychological study of programming," *Comput. Surveys*, vol. 13, pp. 101-120, Mar. 1981.
- [38] B. Schneiderman, *Software Psychology*. Cambridge, MA: Winthrop, 1980.
- [39] M. E. Sime, T. R. G. Green, and D. J. Guest, "Psychological evaluation of two conditional constructions used in computer languages," *Int. J. Man-Mach. Studies*, vol. 5, pp. 123-143, 1973.
- [40] ---, "Scope marking in computer conditionals—A psychological evaluation," *Int. J. Man-Mach. Studies*, vol. 9, pp. 107-118, 1977.
- [41] H. A. Simon, "The architecture of complexity," in *Proc. Amer. Philosoph. Soc.*, vol. 106, pp. 467-482, Dec. 1962.
- [42] W. J. Tracz, "Computer programming and the human thought process," *Software-Practice and Experience*, vol. 9, pp. 127-137, 1979.
- [43] I. Vessey, "An empirical study of some factors affecting program development," Univ. Queensland, unpublished, Nov. 1981.
- [44] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: An empirical study," *Commun. Ass. Comput. Mach.*, pp. 128-134, Feb. 1983.
- [45] C. E. Walton and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, pp. 54-73, 1977.
- [46] L. M. Weissman, "A methodology for studying the psychological complexity of computer programs," Ph.D. dissertation, Univ. Toronto, unpublished, 1974.
- [47] E. Yourdon, *Techniques of Program Structure and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [48] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.



Iris Vessey received the M.Sc. degree, The Diploma of Information Processing, and the M.B.A. degree from the University of Queensland, St. Lucia, Qld., Australia, in 1965, 1971, and 1977, respectively.

Since 1973 she has been Lecturer in Information Systems at the University of Queensland. Her research interests include software design and maintenance, and the nature of problem solving expertise in computer programming and debugging.



Ron Weber received the B.Com. degree from the University of Queensland, St. Lucia, Qld., Australia, in 1970 and the M.B.A. and Ph.D. degrees from the University of Minnesota, Minneapolis, in 1975 and 1977, respectively.

Since 1981 he has been Professor of Commerce at the University of Queensland. His research interests include computer control and audit and structured techniques as a manifestation of a general theory of complexity applied to artifacts.

# Use of Software Engineering Practices at a Small MIS Shop

TERRY C. SNOW

20

*Abstract*—This paper describes the software engineering practices used by the MIS Department at United Technologies Microelectronics Center (UTMC). It describes the life cycle of a software change and the controls established to implement the change. Several software tools used by UTMC to develop and control software development and integration are described as well as methods used to integrate vendor software packages with in-house developed software. A computerized system for tracking and controlling users' modification requests has been developed. Software requirements for a small MIS shop are examined and compared to the requirements for large software development projects.

## I. INTRODUCTION

FOR some time, large software development projects have used the latest software engineering practices to improve productivity and reduce errors. Companies including SDC, Ford Aerospace, and TRW are involved with large software projects and have developed software engineering practices that provide configuration control, integration of multiple software releases, and independent test and validation functions. Now these same software engineering practices are being applied at a small MIS shop within the United Technologies Microelectronics Center (UTMC) in Colorado Springs, CO.

## II. UTMC BACKGROUND

UTMC provides integrated circuits for use by other companies within the United Technologies Corporation (UTC). UTMC allows the UTC companies to design the circuits using an internally developed CAD tool and provides the back-end manufacturing for die produced from a wafer manufactured by Mostek, a sister UTC company.

UTMC hardware configuration consists of three VAX-780's, two VAX-750's, and one VAX-730 connected by Ethernet. Most UTMC employees have terminals in their offices, many of which can be connected to any VAX via Ethernet. In addition, Decnet lines connect the UTMC computers with computers at Mostek, Carrollton, TX. The software configuration for the MIS section is shown in Fig. 1. Business software consists of accounts payable, payroll, general ledger, budget, personnel, document control, marketing, and project control software packages totaling over 500 000 lines of Cobol and Datatrieve code. Manufacturing software consists of work-in-progress, engineering data collection, engineering analysis, and activity planning systems totaling over 1.2 million lines of Cobol and Macro code. Data files consume approximately 500 000 blocks (256 million bytes of information).

Manuscript received August 1, 1983.

The author is with the United Technologies Microelectronics Center, Colorado Springs, CO 80907.

## III. NATURE OF A SMALL MIS SHOP

The small MIS shop which provides business operational support differs from a large software development project in several key features. The small MIS shop tends to satisfy software requirements by purchasing a large number of off-the-shelf software packages and integrating them. This provides a large diverse library of software systems without needing to maintain a large development staff. Thus, the small MIS shop tends to be more of an integrator of software packages than a developer. Software development constitutes enhancements to off-the-shelf and contractor-developed packages so they meet company requirements. The emphasis is on producing a reliable product instead of high-volume production rate.

The large library of software together with the necessary tailoring and/or enhancements to meet correctness requirements dictate a need for

- configuration control
- control of software releases
- separate integration of production and test systems
- independent test and validation of all software enhancements.

Because large software development projects have the same basic requirements, similar software engineering practices can be applied both to large and small MIS shops.

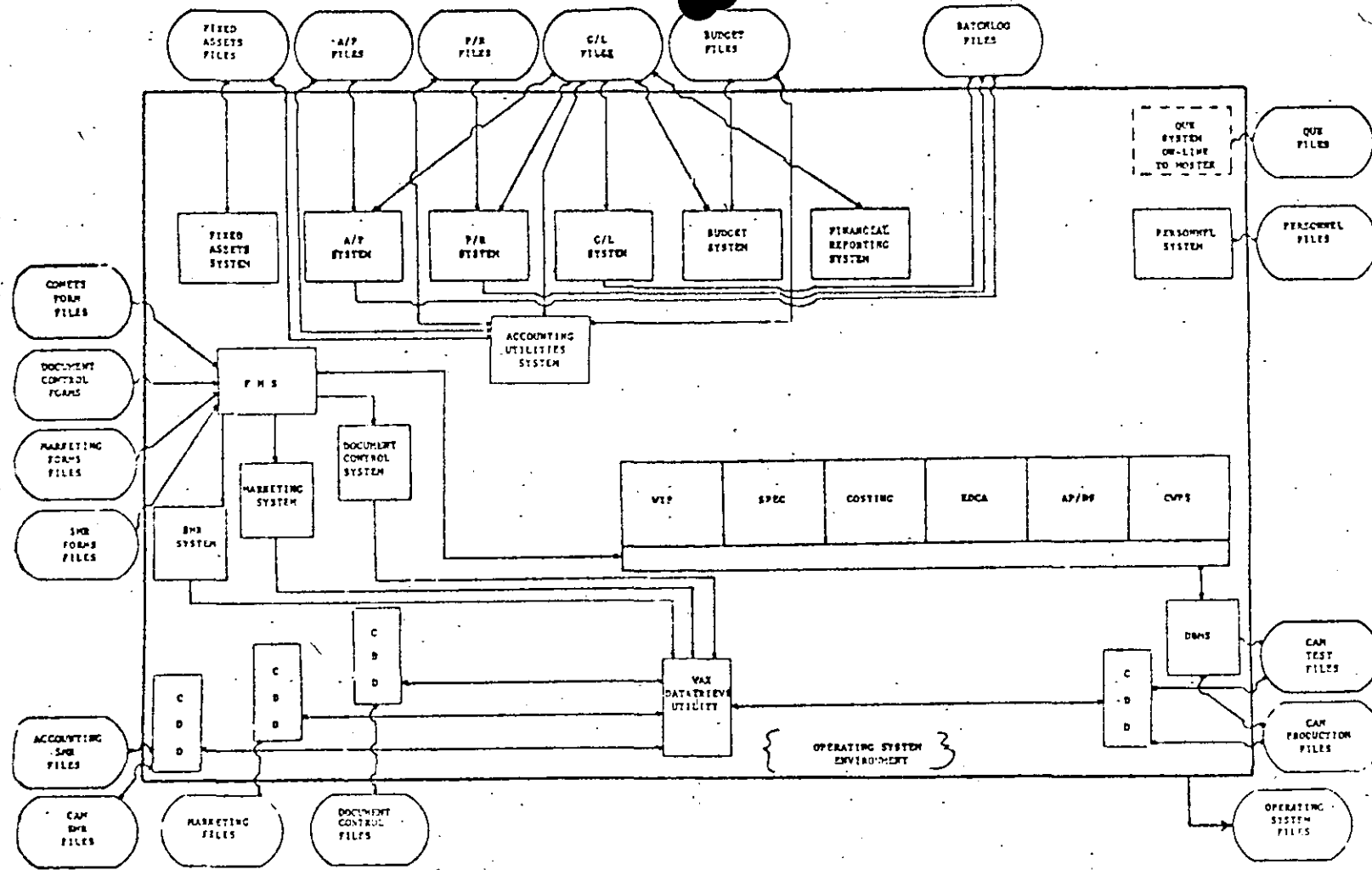
## IV. SOFTWARE ENGINEERING PRACTICES AT UTMC

UTMC's MIS section is responsible for satisfying the software requirements for several departments: Finance, Personnel Resources, Communications, Manufacturing, Technology Development and Assurance, Planning and Administration, and Customer Support. In order to prioritize work, provide status information on outstanding projects, and ensure the reliability of all modifications, the MIS section developed a set of software engineering practices. The users of the software systems accepted these practices very well because the system provided a method of requesting changes and tracking the progress of their requests.

UTMC's software engineering practices emphasize system correctness while providing configuration control. The goals are to provide systems that satisfy the user requirements and ensure the reliability of the finished products. The company's MIS section controls the integration of many diverse software packages into the company's total software requirements.

Programmers and engineering groups have separate responsibilities at UTMC. Two three-person programming teams, each with a lead programmer, are responsible for the design and





21

A/P - ACCOUNTS PAYABLE  
 P/R - PAYROLL  
 G/L - GENERAL LEDGER  
 CDD - COMMON DATA DICTIONARY

WIP - WORK IN PROCESS  
 EDCA - ENGINEERING DATA COLLECTION  
 AP/DS - ACTIVITY PLANNER/DISPATCH  
 CWPS - COMPANY WIDE PLANNER

Fig. 1. MIS software architecture.

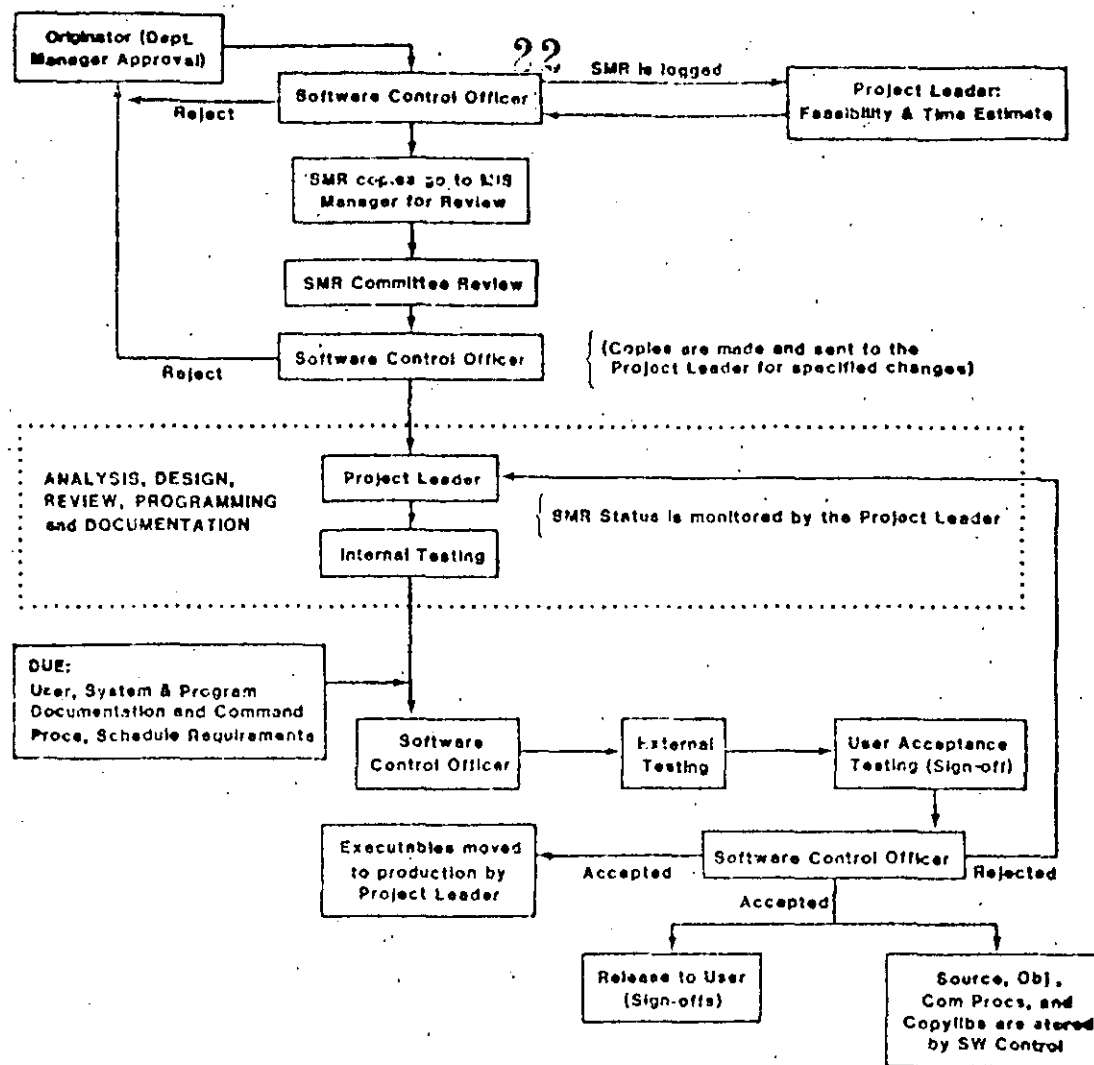


Fig. 2. Software control cycle.

coding of all software enhancements. The three-person system engineering group defines user requirements, provides software integration control and software integration, provides quality control validation of software, and controls and maintains the database.

The software control cycle is illustrated in Fig. 2. The cycle begins with a user submitting a request to the MIS section and ends with final acceptance of the completed change by the user.

Since the MIS section supports the company's software needs, heavy interaction with the users is required. The MIS section has developed a computerized system that allows users to submit system modification requests (SMR's) to the MIS Department (see Fig. 3 for SMR form). The SMR tracking system allows complete tracking of an SMR from submission through production and test. The SMR tracking system is written in Datatrive, a data management language for DEC computers. It consists of over 15 procedures to allow for interactive data input, status displays, and report generation.

The SMR tracking system is accessible both by the MIS section and the users. It gives the users the status of their SMR's, and provides a complete set of reports that allows the MIS section manager to monitor the progress of the group. It also supplies the information to a graphics package (RS1) that

produces five monthly multicolor progress charts for management review (see example Figs. 4 and 5). RS1 was developed by Bolt Beranek and Newman, Inc. to combine basic statistical analysis capabilities with a graphics package to produce bargraphs, histograms, curve fitting, and two- or three-dimensional graphs.

After the SMR is prioritized by a high-level management committee, it is given to programmers who use top-down design and structured programming techniques for design and development. Programmers use structure charts and pseudocode to develop the design and then conduct a review walk-through of the design with their lead programmer before coding begins. All development is performed interactively via computer terminals. The programmers take maximum advantage of the DEC development tools including forms management system (FMS), Datatrive (query language), VAX-11 DBMS, DBQ (DBMS query language), and Cobol.

FMS is a tool used for developing form applications to run on a VT100 terminal. FMS associates constant data with the form, not the application program, resulting in simplified application program maintenance and increased application program flexibility. Datatrive is a data management tool that provides both interactive and program-callable access to data



SMR BACKLOG BY MONTH

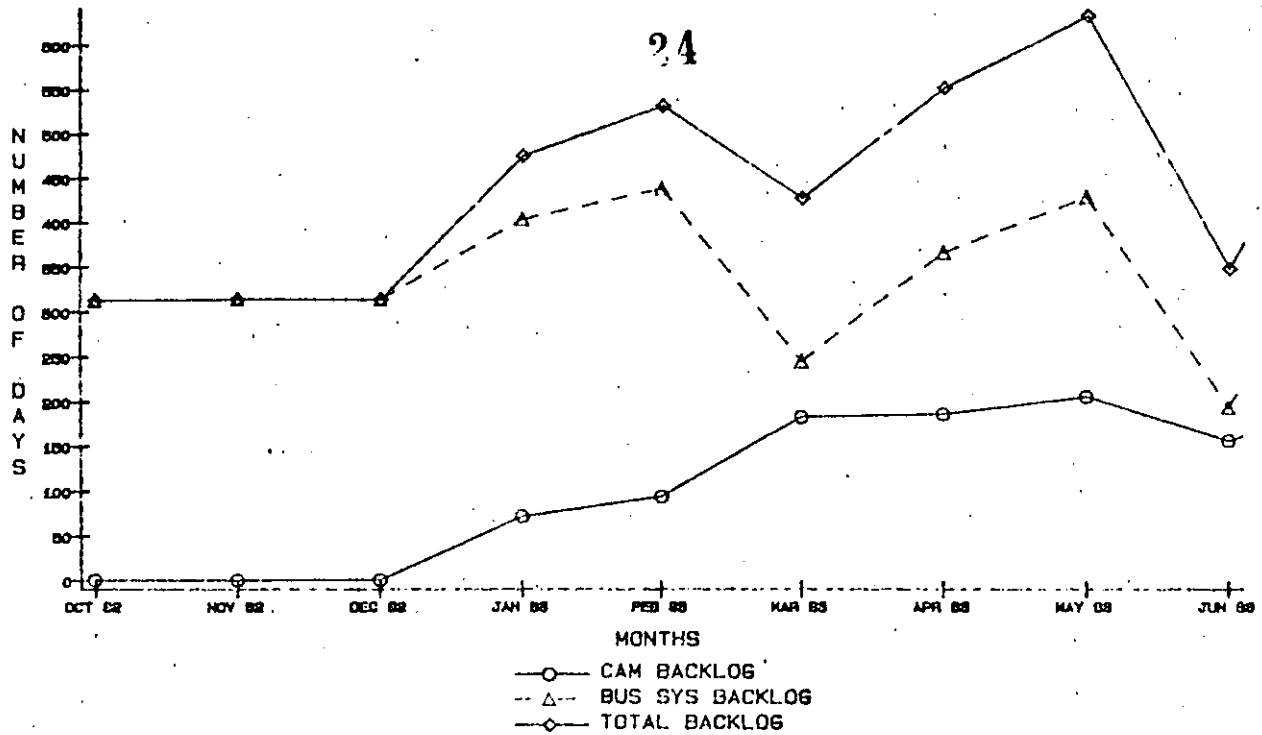


Fig. 4. SMR backlog by month.

SMR STATUS BY DEPARTMENT

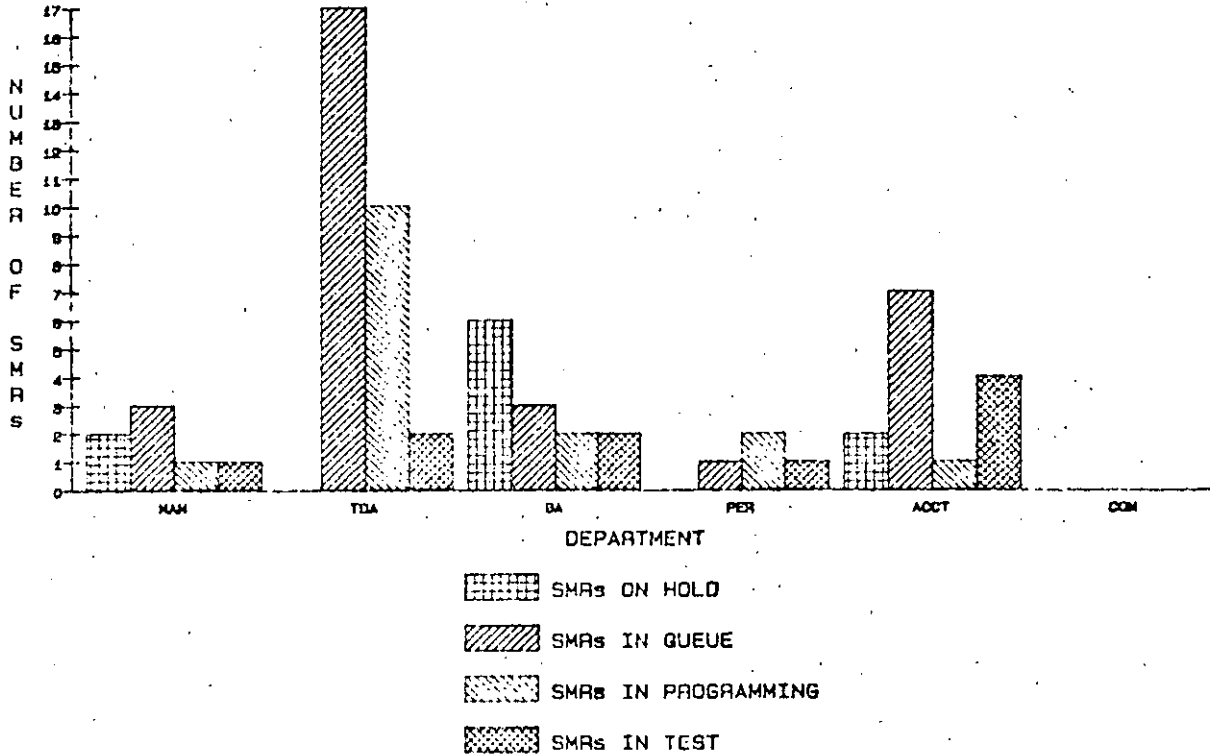


Fig. 5. SMR status by department.

in file organizations. It is a comprehensive query and report writer with full update capabilities. VAX-11 DBMS is a full-scale CODASYL-compliant database management system based on the Working Document of the ANSI Data Definition Language Committee (March 1981). DBQ is used to emulate code structure as a development aid for programmers accessing the database.

The developed or enhanced software is turned over by the programmers to system engineering for independent test and validation. This provides a quality control check for UTM-developed software as well as for off-the-shelf and special contractor-developed software packages. A two-level test which consists of a regression test and a test of the new capability is performed for new releases. After the change is vali-

dated by system engineering, MIS releases it to the user for final acceptance testing.

After the software is accepted, the system engineering group integrates it into the production system. The different software release versions are maintained by a DEC software library system called Code Management System (CMS). CMS allows the operational version of the software to be maintained in a production library while developmental versions are maintained in test libraries. CMS is a program library system that members of a software project use as an aid in program organization, development, and maintenance. It allows project members to retrieve copies of library files, make changes to these copies, and then replace them in the library. It allows more than one project member to work on the same file at the same time without losing any of the modifications. Since UTMC also receives software releases from vendors to the same modules that UTMC programmers are modifying, CMS is used to merge modifications into a single source module containing both sets of changes.

Release versions of the software are built with the aid of the Module Management System (MMS). MMS is a tool that automates building software systems via linkage commands. MMS interfaces with CMS to look for elements in the CMS library. MMS determines the components of a software system that have changed and updates only those components. CMS and MMS also provide security controls for code access/replacement.

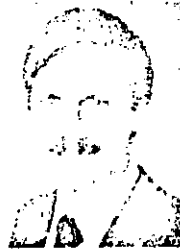
Since UTMC uses many software packages developed by outside vendors, many of these same engineering practices are applied to the vendor software; users are involved with both defining requirements and final acceptance testing; all software releases are thoroughly tested by MIS before release to production; and the SMR system is used to track and report errors to the vendors. Vendor supplied software must be compatible with the existing UTMC VAX architecture which includes layered products such as DBMS, Datatrieve, Forms Management Systems, and Common Data Dictionary. Modular system design of vendor supplied software is a key selection criterion as well. Product packaging for UTMC purchased systems must include internal and external specifications, user documentation, source code, executables, linkage commands, data conversion routines, and all supporting job control procedures required for system implementation and operational usage.

Because the product selection criteria are strictly enforced, the time requirements for implementation and integration are minimized. Minor modifications to linkage modules and job control code are usually required to customize the software package for operational use. On the average, it takes three days to install a major software release from a vendor into the test system. It takes one to three weeks to thoroughly test the package prior to implementation of a fully operational system in production.

Each month, the UTMC MIS shop processes over 30 user requests varying in scope from one day to several months for completion. Use of the above control and development procedures, allows for

- 1) complete knowledge of current status of all requests
- 2) a quality product that is well tested
- 3) user interaction in defining requirements.
- 4) user acceptance of completed products.

Thus, UTMC has demonstrated through these procedures that many of the software engineering practices currently used by leading software development companies—configuration control of software releases, integration of software into production and test, and validation of software—can also apply to small MIS shops.



Terry C. Snow received the B.S. and M.S. degrees in mathematics from the University of Oklahoma, Norman, OK, in 1970 and 1971, respectively.

From 1972 to 1976 he was a Computer System Analyst for the United States Air Force. He worked on the Space Defense Center Computer System for the North American Aerospace Defense Command. From 1976 to 1981 he worked for the System Development Corporation as the Space Computational Center Software

Manager responsible for proposals for new software systems. He supervised software development for man-machine interface, operating systems, and application development. In February 1981 he joined the United Technologies Microelectronics Center to establish their MIS Department. As MIS manager he is responsible for business and manufacturing software systems.

Mr. Snow is a member of the Phi Eta Sigma, Phi Mu Epsilon, the National Honor Society, and is an affiliate member of IEEE Computer Society. He received the Air Force Commendation Medal, North American Air Defense Command Certificate of Achievement, and Certificate of Recognition for Outstanding Technical Support to the Space Defense Center.

2 much had been attempted at one time. Accordingly, the instruction continued in 1974, concentrating on structured programming, top-down programming, structured walk-throughs, and program design language, the latter being the subject of this paper. As a result of this further effort, use of these four techniques has become a stated policy at MCAUTO. This investigation into the other techniques continued in 1975, with Hierarchy plus Input Process-Output (HIPO) also showing great promise as a system analyst tool.<sup>4</sup>

The logic specification standards that have been used at MCAUTO are roughly equivalent to detailed flowcharts, in which numbered English sentences are substituted for the various flowchart symbols. As detailed flowcharts were formerly used to create the required detail, so were logic specification standards. Although flowcharts and logic specification standards proved adequate for smaller and less complex applications, it was recognized in the early 1970s that more complex applications are correspondingly more difficult to describe and specify by the use of flowcharts. That increasing size and complexity of applications had gradually outgrown the capability and scope of earlier logic specifications was the primary condition that set the stage for the new technique of using a program design language.

### **Program design language**

The program design language that is presented in this paper is a tool for designing programs in detail prior to coding. At MCAUTO, the program design language is used both as a language and as a program development methodology. The program design language is syntactically simple and supports structured control figures<sup>5</sup> tailored for PL/I and COBOL. The syntax of the language is described in the Appendix. Top-down program development methodology and elements of stepwise refinement<sup>6</sup> and levels of abstraction are used with the program design languages.<sup>7</sup> The methodology is described in this paper. At MCAUTO, programmers use the program design language in conjunction with structured walkthroughs, top-down implementation,<sup>8</sup> and structured programming.<sup>3,4</sup> Although the value of the program design language has not been evaluated apart from the other techniques, the language is believed to be a major contributor to increased productivity.

The program design language, as a form of pseudocode, has the following characteristics:

- Notation is used to state program logic and function in an easy-to-read, top-to-bottom fashion.
- It is not a compilable language.

- It is an informal method of expressing structured programming logic.
- It is similar to a programming language (such as COBOL or PL/I), but is not bound by formal syntactical language rules.
- Conventions exist that pertain to the use of structured figures and indentation to aid in the visual perception of the logic.
- The primary purpose is to enable one to express ideas in natural English prose.
- The language permits concentration on logical solutions to problems, rather than the form and constraints within which the solutions must be stated.
- The language uses flowchart replacements, program documentation, and technical communication at all levels.
- Program design is expressed readably, and can be converted easily to executable code.

The program design language was initially used to teach structured programming to the programmers. As a teaching aid, the language helped the programmers make the transition to thinking in terms of a hierarchy of routines that consisted of basic structured figures.<sup>4</sup> When programmers started to implement application systems using flow charts and other earlier methods in which the programs were of the nonhierarchical and nonstructured type, the refining process included making hierarchical and structured program designs. Using the program design language rather than structured flowcharts or structuring the standard logic specifications proved to be the easiest way to improve the program design. Continued use and refinement of the language has established it as the medium of choice for either creating or refining a detailed program design. Although more experience with HIPO is needed, it presently appears that HIPO<sup>4</sup> may become the medium of choice for system design, and further become an excellent input for detailed program design. In time, HIPO may be as useful to analysts as the program design language is to programmers.

#### Top-down program design

Simplicity is a key attribute to the program design language syntax, conventions for which are given in the Appendix. In general, when the language is written according to the guidelines to be discussed in this paper, statements in the language are easy to transform into programs. More importantly, the simplicity frees the designer, who is usually a programmer, to concentrate on developing the detailed logic of a program. While the systematic application of the program design language facilitates program design, the language is not a simplistic means of doing the whole job of programming. Detailed program design is an iterative process, with the possibility that details discovered in the later

stages of design may lead to modifications in previous portions. Although experience in using the language and familiarity with the application may reduce the impact of such incidents, one should usually plan to complete a detailed design before starting to code a program. Since the program design language is easier to change (or rewrite) than actual code, cleaning up a program design in that language is usually more cost-effective than cleaning up program code. The primary objective in defining a procedure for the systematic application of the program design language is to provide a general scheme of things to be done during detailed program design.

The systematic application of the language is to apply the principles of top-down programming to the detailed program design function, which we term "top-down program design." This implies that the process of program design can be described as a hierarchy of discrete functions, which further implies that the work product (the program design) should be a hierarchy of discrete units that ease program implementation in a top-down manner.

According to program design language conventions, the discrete units in the case of program design are one-entry-one-exit routines (as in structured programming) that are no larger than one page. In most cases, all the detailed logic for a program does not fit on one page, a fact that leads to a squeezing down of detail into lower-level routines, and results in a number of hierarchically related routines. The syntax and conventions of the program design language promote a program design that meets the objectives of top-down programming. An example that shows the squeezing of detail into lower-level routines and the formation of hierarchically related routines is given in the following section.

### **Top-down program design example**

The top-down design process may be regarded as having the following three distinct phases:

- Determining requirements.
- Abstracting functions.
- Expanding functions.

Obviously, the time and effort needed for each of these phases depends on the designer's experience and ability. Likewise, the particular way in which the functions are designed depends on the amount and organization of the source information. If the source data for a program design do not include completed file designs, report layouts, and user input definitions, then the application system design is not ready to be expanded into a detailed



program design. Moreover, a system design should include, as necessary, functions that the program should perform and any constraints on the program (such as field edits or sequences of calculations). Even after assuming that one has at least the minimum system-level specification for the program, there may still be wide variations in the level and volume of details and in the organization of those details. The optimum system specification is a hierarchy of user-oriented functions that includes only those details that are directly related to a user's requirements.

5

The establishment of practical guidelines for the optimal level of detail and organization for system-level specifications requires the active cooperation of both analysts and programmers. Whether done by analysts or programmers, the following three basic functions of detailed program design must still be performed: determine the requirements, abstract the functions, and expand the functions.

At the time of a detailed program design, the determining of program requirements consists primarily of studying the system specifications for the program. Any items that are vague, missing, undefined, or contradictory should be clarified before plunging into detailed program design. If the system specifications do not, at some point, provide a simple statement of user requirements, then write down such items as they become apparent. This point is crucial because the abstractive process should be in terms of the user's requirements. Likely sources are the definitions of output reports, files, screens, etc. The report specifications for a simple report generation program might yield the following functions:

determining  
requirements

- Accumulate total sales for each salesman.
- Accumulate total sales for each district.
- Accumulate total sales for all districts.

Examination of the input specification for the program might reveal the following constraints:

- The sales file has only one kind of record.
- Each sales record includes salesman name and number, and district number.
- Sales records are in order by salesman identification within each district.
- There may be several sales records for a salesman.

Additional constraints, such as "skip to new page after printing a district total" might be found.

If it is assumed that the specifications at the source specification level of detail do not express the user's requirements, the objec-

tive is to build a complete list at this level. It is not necessary to organize the list. Rather, one should concentrate on discovering all the functions that the user wants to be performed. Assuming this criterion, one might reasonably eliminate all the previously listed functions and constraints except the following:

- Accumulate total sales for each salesman.
- Accumulate total sales for each district.
- Accumulate total sales for all districts.
- Skip to new page after printing a district total.

At this point, a discussion with the analyst or user might be profitable. In any case, the requirements should be thoroughly understood, so that abstracting the functions—which is discussed in the following section—may be started.

**abstracting  
functions**

Abstracting the functions consists of discriminating between functions that are subfunctions and those that are main functions. To begin abstracting the functions, one first decides whether there is one function in the list that implies all the others. If there is none, then the programmer invents such a comprehensive function (i.e., he abstracts a general statement). For example, the report program function might be to "Summarize Sales," which implies that all the other sales functions are subfunctions. In that case, what are the relationships among the five functions on a main and subfunction basis? A good starting point for decision making is to organize the list by grouping all functions that have related inputs or outputs and by ranking each group in a most-general-to-most-detailed order. Since the report program has only one input file and one output report, grouping is not necessary. Ranking the sales functions yields the following general-to-detailed list:

1. Accumulate total sales for all districts.
2. Accumulate total sales for each district.
3. Skip to new page after printing district total.
4. Accumulate total sales for each salesman.

It appears that 2 and 3 are at the same functional level; that is, 1 implies 2 and 3 implies 4. This relationship suggests some minor reordering, which is brought out by the following list:

1. Accumulate total sales for all districts.
2. Accumulate total sales for each district.
4. Accumulate total sales for each salesman.
3. Skip to new page after printing district total.

Compare the new list with the report layout and note that there is a good match-up, especially if the basic functions are expand-

ed to designate the various totals that are to be printed as follows:

- A1. Accumulate total sales for all districts.
- B1.     Accumulate total sales for each district.
- C1.             Accumulate total sales for each salesman.
- C2.             Print total sales for each salesman.
- B2.     Print total sales for each district.
- B3.     Skip to new page after printing district total.
- A2. Print total sales for all districts.

7

At this point, the following three functional levels have been identified: all districts, each district, and each salesman. Each functional level contains a mixture of relatively simple functions, e.g., print and skip; and more general functions, e.g., accumulate. Generally, one cannot code a program from abstractions of function at this level. Definitions of the too-general functions must be expanded until all functions are sufficiently defined.

The expanding of functions consists of repeating the following four basic steps until all functions in the design have been sufficiently simplified to be coded: selection, analysis, specification, and verification. The appropriate point at which to stop depends on a programmer's familiarity with the program design language, structured programming, and the functions. Usually, the greater a programmer's experience with the program design language, the higher will be the level of detail that he uses. That is, when a programmer first starts using the language, more detailed definitions are needed (and written) than are needed after he has become accustomed to using the language. If a next lower level of expansion of named functions results in program design language statements that are program code, then the current level of expansion is probably sufficient. Of course, if all the statements can already be transferred into code on a one-for-one basis, the design is complete.

expanding  
functions

Selecting a function is the first step in expanding the functions. Expansion should generally be accomplished in a top-down manner. That is, expand the highest level (as yet undefined) function next. When faced with a choice of undefined functions at the same level, the main-line, or most important function, is usually expanded first. In the program example used in this paper, the function labeled A1 is the natural candidate for being expanded first. Since the expansion of A1 may produce another function that needs expansion, it is premature to assert that B1 should be expanded next. After having selected a function, the next step is to analyze it.

Analyzing a function is the process of deciding what must be done to accomplish a given function. This is sometimes referred

to as breaking a function down into subfunctions. In the event that major subfunctions have already been determined, analysis may consist of defining supportive subfunctions. For example, B1, B2, and B3 are major subfunctions of A1. Supportive subfunctions of A1 might be the following:

8

Set total for all districts to zero.  
Add district sales to grand total.

Since A is the highest level in the program, the following data processing functions must also be done:

Open files.  
Close files.

After the subfunctions have been identified, their relationships to one another can be specified.

Specifying relationships of the various subfunctions is accomplished by using the appropriate conditions and structured control figures. Specification may be done by using existing data variables, or it may require the definition of new data variables. New data variables should be noted as such, to facilitate both the eventual coding of a function and the expansion of lower-level functions during design. In effect, subfunctions and their relationships to one another should constitute a complete definition of function. For example, the A level might be specified as follows:

*Summarize sales*

Open files.  
Set total for all districts to zero.  
DO WHILE more sales data.  
    Accumulate total for a district.  
    Add total for district to total for all districts.  
ENDDO  
Print total sales for all districts.  
Close files.

In this example, the statement "Accumulate total for a district" refers to the B- and C-level functions. We, therefore, proceed with the selection, analysis, and specification of the B- and C-level functions.

*Accumulate total for a district*

Set total for a district to zero.  
Set current district to district in sales record.  
DO WHILE current district matches district in sales record.  
    And more sales data.  
    Accumulate total for a salesman.

Add total for a salesman to total for a district.  
ENDDO  
Print total for a district.  
Skip to a new page.

9

*Accumulate total for a salesman*  
Set total for a salesman to zero.  
Set current salesman to salesman in sales record.  
DO WHILE current salesman matches salesman in sales record:  
    And current district matches district in sales record.  
    And more sales data.  
    Add sales data to total for a salesman  
    Read sales record  
ENDDO  
Print total for a salesman.

A programmer who is experienced in structured programming should find the specification and expansions just given relatively easy to code. Although some of the loop conditions have only been named (e.g., more sales data), their expansion into code should not pose a great problem. Before doing any coding, however, a little desk checking is often found to be of value.

Seldom can practical programs be completely defined on a single page using the program design language. More likely, the first page of material that is written in that language names the functions that are to be expanded on another page. The first- (or highest-) level page of program design language statements defines the environment of the lower-level function. After the completion of one page in that language, it is often useful to take a checkpoint and verify the completeness and correctness of a function that is defined by the program design language. In doing the verification, it may be helpful to list the various combinations of inputs needed to test a routine, in effect, to define—at least in part—what must be done to test the program. In any event, one last thorough examination of a unit of design description before proceeding to lower-level design or coding may save subsequent rework. For example, attempting to process even one record by the example report program reveals the need for a read-sales-record statement before the first DO WHILE at the highest level, i.e., *Summarize sales*.

verification

### Experience and conclusions

At MCAUTO, the following major advantages of using the program design language instead of traditional techniques for detailed program design have been observed:

- Ease of writing programs;

- Ease of changing programs.
- Transferability into structured code in a top-down manner. 10
- Ease of reading programs, especially by nonprogrammers.

The readability aspect contributes to the effectiveness of structured walkthroughs for nonprogrammers. Since the program design language is inherently hierarchical and structured, it also contributes to the success of top-down development and structured programming. Although further experience is needed, it appears that the functional orientation of HIPO also lends itself to expansion into the program design language. Thus the use of the language contributes to the successful use of the other programming techniques.

The systematic application of the program design language is not a cookbook checklist for designing programs. In practice, the individual steps—especially those involved in expanding a design—tend to be done simultaneously, rather than sequentially. Initially, the program designer may be slowed down by his unfamiliarity with manipulating DO WHILEs and IF THEN ELSEs to accomplish his purpose without recourse to GOTOS. With experience, program designs are usually created more readily than otherwise. The resultant designs are typically of better quality than traditional program designs. The better quality of programs designed using the program design language is reflected in relative ease of implementation and maintenance, and by the absence of production errors.

#### ACKNOWLEDGMENTS

The author extends his thanks and appreciation to the MCAUTO programmers for their interest and perseverance during our mutual learning period. He especially thanks Charles E. Holmes (MCAUTO St. Louis), John E. Hiles (MCAUTO West), and E. Jean Bland (IBM, St. Louis) for the imagination, dedication, and leadership that contributed to the successful adaptation of the methods discussed in this paper.

#### CITED REFERENCES

1. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* 11, No. 1, 56-73 (1972).
2. C. E. Holmes and L. W. Miller, *Proceedings, 37th Meeting of GUIDE International*, Boston, Massachusetts, October 28-November 2, 1973 (560-575).
3. C. E. Holmes, *Proceedings, 39th Meeting of GUIDE International*, Anaheim, California, November 3-8, 1974 (689-700).
4. *HIPO—A Design Aid and Documentation Technique*, Order No. GC20-1851, IBM Corporation, Data Processing Division, White Plains, New York 10604.
5. *Improved Programming Technologies—An Overview*, IBM Systems Reference Library, Order No. GC20-1850, IBM Corporation, Data Processing Division, White Plains, New York 10604.

6. N. Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1973).
7. E. W. Dijkstra, "The structure of T.H.E. multiprogramming system," *Communications of the ACM* 11, No. 5, 341-346 (1968).

**Appendix**

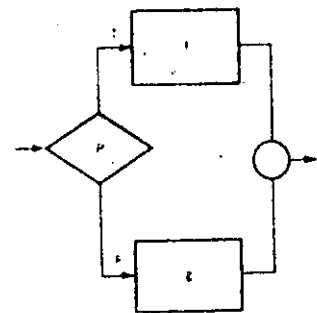
The syntax of the program design language includes provisions for expressing the three basic logic constructs (or figures) of structured programming: SEQUENCE, IF THEN ELSE, and DO WHILE. In the program design language, these constructs have been augmented with the PERFORM UNTIL and CASE constructs. Each logic construct has a definite and simple syntax. In addition to the statement syntax, conventions have been established for the use of indentation and the size of self-contained units of the program design language. The SEQUENCE construct is used to describe any action or work that is followed by the next sequential construct. In control structure forms, SEQUENCE is represented by the function of a subroutine block as shown in Figure 1, where f is the action or work to be done. Syntactically, SEQUENCE represents a simple English sentence, with at least a verb and an object. In practice, the language is most meaningful when action-oriented statements with objects that are natural to the problem are used. Compare, for example, the following sentences: "Print," with "Print XYZ," and with "Print gross sales for salesman."

Figure 1 Control structure for the SEQUENCE logic construct



The IF THEN ELSE construct is used to describe binary decisions. In its most general form, that logic construct is used to describe the conditions under which one of two actions are to be taken. The control structure for IF THEN ELSE is given in Figure 2. The symbol is the predicate (or list of conditions), and f and g are alternative actions. Note that f and g may include any of the logic constructs, and are not limited to being the SEQUENCE construct. The general syntax of the IF THEN ELSE construct is as follows:

Figure 2 Control structure for the IF THEN ELSE logic construct



```

IF      P
THEN    f
ELSE    g
ENDIF
  
```

The IF, THEN, ELSE, and ENDIF should always be vertically aligned and displayed in all capitals for ease of reading. When p consists of multiple simple conditions, each condition should be written on a separate line, and all conditions should be vertically aligned, as, for example, in the following way:

```

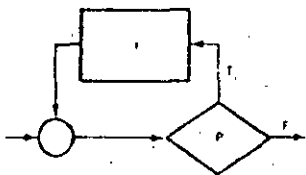
IF      No more data or
        Different department.
THEN    Print total department sales.
ELSE    Add sale amount to total department sales.
ENDIF
    
```

The IF and ENDIF conditions are required. When, however, either the THEN or the ELSE clause is not needed, they may be omitted. In other words, the following are syntactically valid forms of the IF THEN ELSE logic construct.

```

IF      p
THEN    f
ENDIF
and
IF      p
ELSE    g
ENDIF
    
```

Figure 3 Control structure for the DO WHILE logic construct



The DO WHILE logic construct is used to describe the repetition of an action under prescribed conditions (looping). The control structure for DO WHILE is shown in Figure 3, where p is the predicate (or list of conditions) and f is the action to be taken. (Note that Figure 3 is a decision loop in which the action is taken when a condition is true.)

The program design language syntax of the DO WHILE construct is as follows:

```

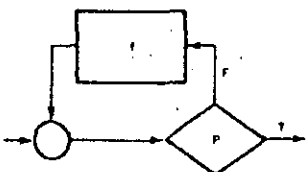
DO WHILE  p,
          f
ENDDO
    
```

where the DO WHILE and ENDDO conditions are vertically aligned and capitalized. Consider the following pseudo code sequence that is based on the example in the body of this paper:

```

DO WHILE  More data and
          Same district:
          Accumulate district sales total.
          Read next sales record.
ENDDO
    
```

Figure 4 Control structure for the PERFORM UNTIL logic construct



The PERFORM UNTIL construct is used to describe looping, when COBOL is the target language for implementation. Control structure for PERFORM UNTIL is shown in Figure 4, where p is the predicate, and f is the action to be taken. PERFORM UNTIL differs from the DO WHILE in that the PERFORM UNTIL loop exits when p is true, rather than when p is false. In effect, DO WHILE p is equivalent to PERFORM UNTIL not p. By using a PERFORM



UNTIL in the program design language.  $p$  may be written exactly as it is written in COBOL, thus avoiding the errors that might occur in doing a Boolean inversion of  $p$  from the DO WHILE of the program design language to the PERFORM UNTIL of COBOL. The program design language syntax of the PERFORM UNTIL logic construct is given as follows:

```
PERFORM UNTIL    p
                f
ENDLOOP
```

where the PERFORM UNTIL and ENDLOOP are vertically aligned and capitalized. An example fragment taken from the text and expressed in the program design language is as follows:

```
PERFORM UNTIL    No more data or
                  Different district:
                  Accumulate district sales total.
                  Read next sales record.
ENDLOOP
```

In comparing this fragment with the DO WHILE example, note that the loop conditions have been inverted.

The CASE logic construct is used to simulate a branch table. In the appropriate situation, CASE can be an efficient and effective alternative to multiple levels of nested IF THEN ELSE statements. This construct may be applicable when one of  $n$  functions is to be executed, depending on the value of a single variable. The control structure for the CASE construct is shown in Figure 5A. Figure 5B is the IF THEN ELSE logical equivalent of the CASE construct.

The program design language syntax of the CASE construct is given as follows:

```
CASE    variable    OF
        Value 1:    f1
        Value 2:    Value 3:    f2
        Value 4:    f3
        .
        .
        Value n:    fn
ENDCASE
```

Here, "variable" is the variable to be checked for the various "values," and "value  $i$ " is a specific value of the variable to associate with the execution of the function  $f_i$ , which appears on the same line. Note that there may be more values  $n$  than there

synthesize information. A convention has, therefore, been adopted. Simply stated, the convention is that a single unit should not exceed one page of standard  $8\frac{1}{2} \times 11$  inch paper. Furthermore, each logic construct should end on the same page on which it begins. In practice, this results in a package of one-page units where voluminous nested functions are represented by simple names — where they are used — that are then defined in detail on separate pages. Essentially, the program design consists of a number of subroutines that are hierarchically related.

Reprint Order No. G321-5032.

16

# Software Engineering

BARRY W. BOEHM

17

Reprinted from *IEEE Transactions on Computers*, Vol. C-25, No. 12, December 1976. Copyright © 1976 by The Institute of Electrical and Electronics Engineers, Inc.

**Abstract**—This paper provides a definition of the term “software engineering” and a survey of the current state of the art and likely future trends in the field. The survey covers the technology available in the various phases of the software life cycle—requirements engineering, design, coding, test, and maintenance—and in the overall area of software management and integrated technology-management approaches. It is oriented primarily toward discussing the domain of applicability of techniques (where and when they work), rather than how they work in detail. To cover the latter, an extensive set of 104 references is provided.

**Index Terms**—Computer software, data systems, information systems, research and development, software development, software engineering, software management.

## I. 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].<sup>1</sup> 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 and 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 is 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.

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.

## 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.

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)

Manuscript received June 24, 1976; revised August 16, 1976.

The author is with the TRW Systems and Energy Group, Redondo Beach, CA 90278.

<sup>1</sup> Another trend has been added to Fig. 1: the growth of software maintenance, which will be discussed later.

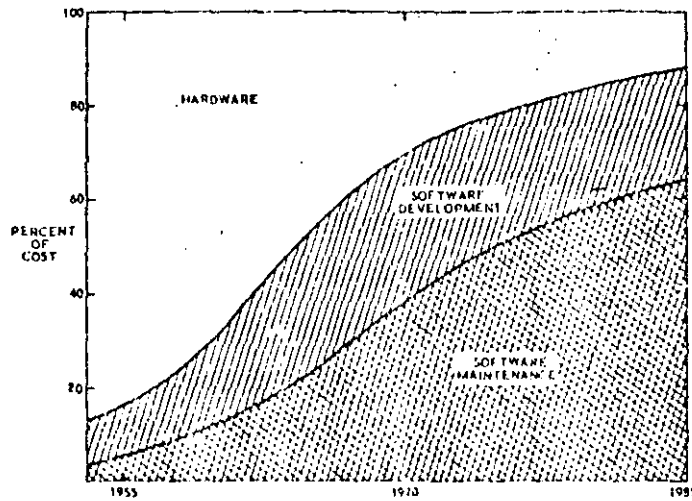


Fig. 1. Hardware-software cost trends.

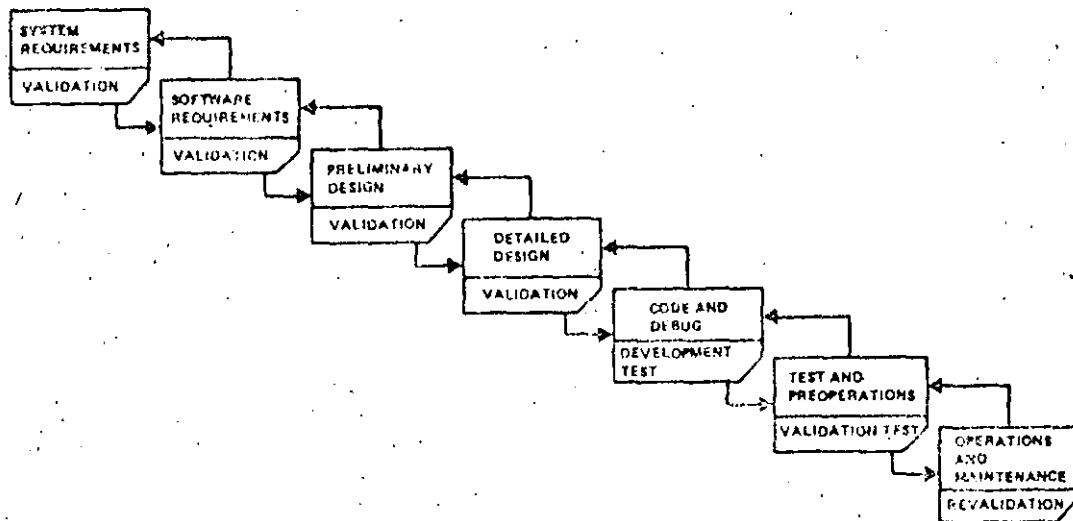


Fig. 2. Software life cycle.

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. Critical 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.

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 designing 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,"

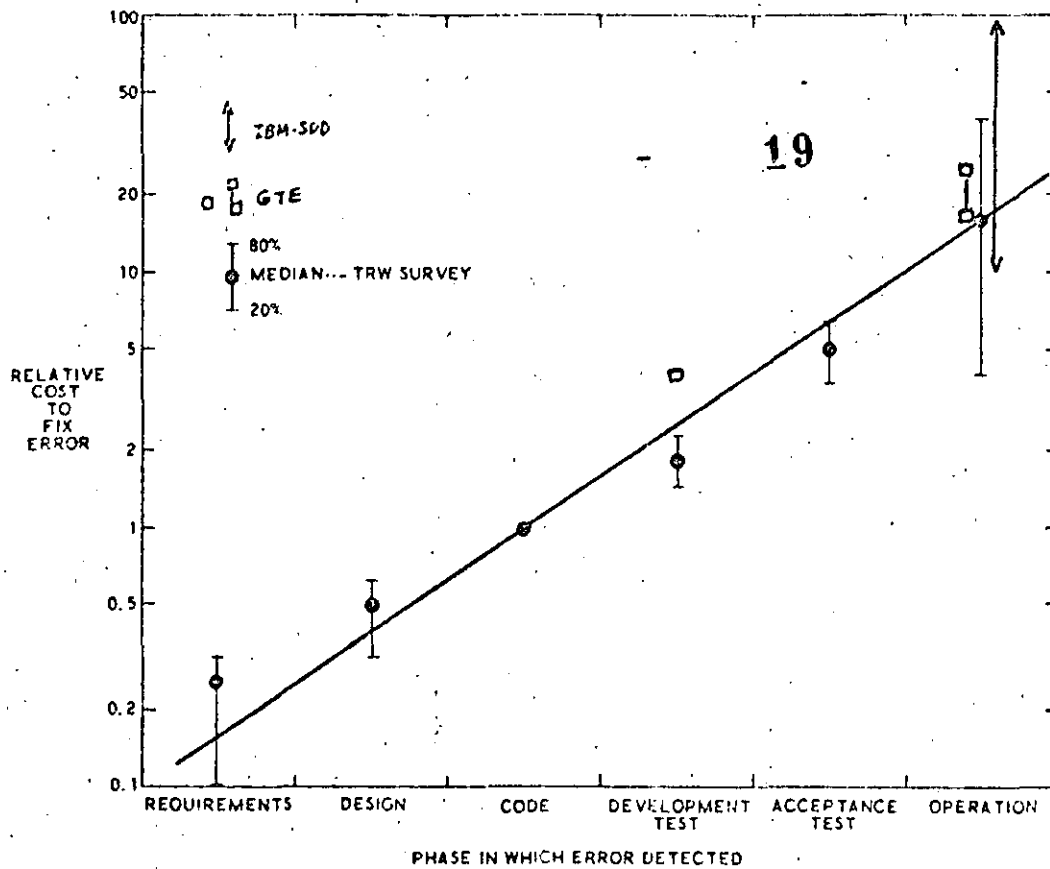


Fig. 3. Software validation: the price of procrastination.

efficient," "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 Peter Broew and his group at the University of Michigan [1]. 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 8 1/2 x 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.

20

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]-[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 currently performed by programmers. If successful, could they drive software 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 am-

biguities, 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 problem space. Even then, though, there will always be 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 I, 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.

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.

##### 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 par-

TABLE I  
Design Degrees of Freedom for New Data Processing Systems  
(Rough Estimates)

21. 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

ticular 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].

##### 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 "out-

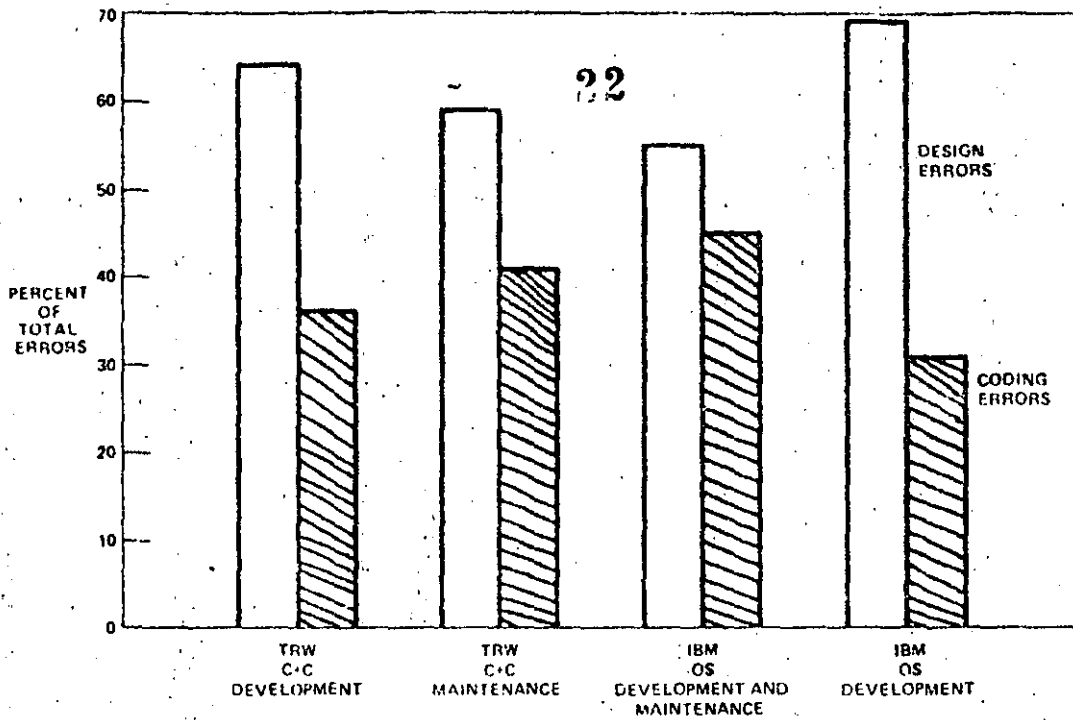


Fig. 1. Most errors in large software systems are in early stages.

put" 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."<sup>2</sup>

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.

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

<sup>2</sup> Problems can arise, however, when one furnishes such a *design choice* 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?



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]-[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], Microw'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.

#### D. Trends

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

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

26  
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 London *et al.* [69] and Luckham *et al.* [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 exercising "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),

- MINIMUM-VARIANCE UNBIASED ESTIMATOR
- PICK  $N$  (SAY, 1000) RANDOM, REPRESENTATIVE INPUTS
  - PROCESS THE 1000 INPUTS, OBTAIN  $M$  (SAY, 3) FAILURES
  - THEN  $R = \text{PROB (NO FAILURE NEXT RUN)} = \frac{N-M}{N} = 0.997$
- OPERATIONAL ESTIMATION PROBLEMS
- SIZE OF INPUT SPACE
  - ACCOUNTING FOR FIXES
  - ENSURING RANDOM INPUTS
  - ENSURING REPRESENTATIVE INPUTS

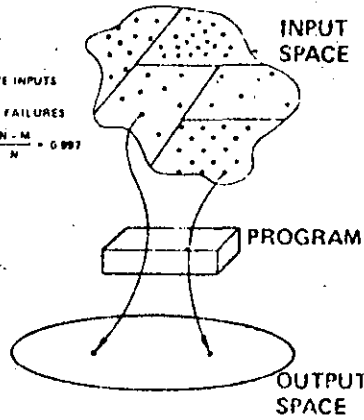


Fig. 5. Input space sampling provides a basis for software reliability measurement.

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].

Despite its size, software maintenance is a highly netted 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

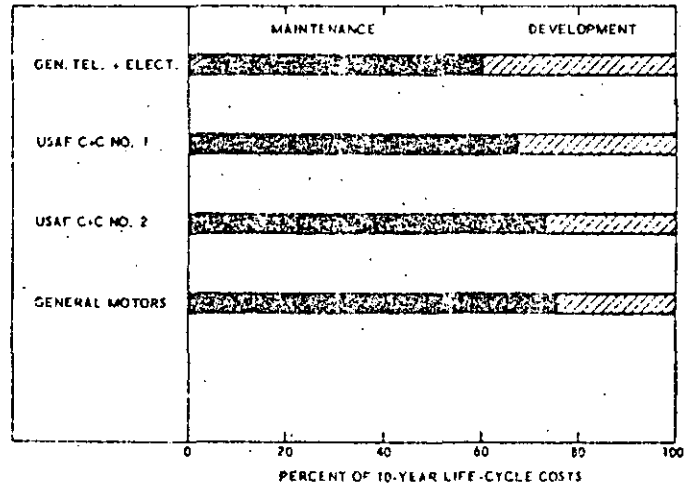


Fig. 6. Software life-cycle cost breakdown.

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 struc-

tures 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].

#### D. Trends

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 issue. 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 novel) 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

[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].

29

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 time 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

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 to date 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  
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 <sup>7</sup> life cycle, e.g., <sup>7</sup> 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.

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.

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*<sup>3</sup> in an *economics-driven* context. And in Area 2, our scientific foundations are so slight that one can seri-

<sup>3</sup> For example, a recent survey of 14 installations in one large organization produced the following profile of its "average coder": 2 years college-level education, 2 years software experience, familiarity with 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 much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who must use it.

ously question whether our current techniques deserve to be called "software engineering."

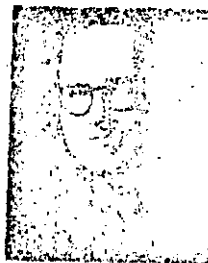
Hardware engineering clearly has available a better scientific foundation for addressing its counterpart of these Area 2 problems. This should not be too surprising, since "hardware science" has been pursued for a much longer time, is easier to experiment with, and does not have to explain the performance 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.

#### REFERENCES

- [1] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48-59, May 1973.
- [2] T. A. Dolotta et al., *Data Processing in 1980-85*. New York: Wiley-Interscience, 1976.
- [3] P. Wegner, "Computer languages," *IEEE Trans. Comput.*, this issue, pp. 1207-1225.
- [4] M. E. Fagan, "Design and code inspections and process control in the development of programs," IBM, rep. IBM-SDD TR-21.572, Dec. 1974.
- [5] E. B. Daly, "Management of software development," *IEEE Trans. Software Eng.*, to be published.
- [6] W. W. Royce, "Software requirements analysis, sizing, and costing," in *Practical Strategies for the Development of Large Scale Software*, E. Horowitz, Ed. Reading, MA: Addison-Wesley, 1975.
- [7] T. E. Bell and T. A. Thayer, "Software requirements: Are they a problem?," *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [8] E. S. Quade, Ed., *Analysis for Military Decisions*. Chicago, IL: Rand-McNally, 1964.
- [9] J. D. Couger and R. W. Knapp, Eds., *System Analysis Techniques*. New York: Wiley, 1974.
- [10] D. Teichroew and H. Sayani, "Automation of system building," *Datamation*, pp. 25-30, Aug. 15, 1971.
- [11] C. G. Davis and C. R. Vick, "The software development system," in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [12] M. Alford, "A requirements engineering methodology for real-time processing requirements," in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [13] T. E. Bell, D. C. Bixler, and M. E. Dyer, "An extendable approach to computer-aided software requirements engineering," in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [14] R. M. Balzer, "Imprecise program specification," Univ. Southern California, Los Angeles, rep. ISI/RR-75-36, Dec. 1975.
- [15] W. A. Martin and M. Bosyj, "Requirements derivation in automatic programming," in *Proc. MRI Symp. Comput. Software Eng.*, Apr. 1976.
- [16] N. P. Dooner and J. R. Lourie, "The application software engineering tool," IBM, res. rep. RC 5434, May 29, 1975.
- [17] M. L. Wilson, "The information automat approach to design and implementation of computer-based systems," IBM, rep. IBM-FSD, June 27, 1975.
- [18] B. W. Boehm, "Some steps toward formal and automated aids to software requirements analysis and design," *Proc. IFIP Cong.*, 1974, pp. 192-197.
- [19] A. B. Endres, "An analysis of errors and their causes in system programs," *IEEE Trans. Software Eng.*, pp. 140-149, June 1975.
- [20] T. A. Thayer, "Understanding software through analysis of empirical data," *Proc. Nat. Comput. Conf.*, 1975, pp. 335-341.
- [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 Trans. Software Eng.*, pp. 125-133, Mar. 1975.
- [22] P. Freeman, "Software design representation: Analysis and improvements," Univ. California, Irvine, tech. rep. 81, May 1976.
- [23] E. L. Glaser et al., "The LOGOS project," in *Proc. IEEE COMP-CON*, 1972, pp. 175-192.
- [24] N. R. Nielsen, "ECSS: Extendable computer system simulator," Rand Corp., rep. RM-6132-PR/NASA, Jan. 1970.
- [25] D. W. Kosy, "The ECSS II language for simulating computer systems," Rand Corp., rep. R-1895-GSA, Dec. 1975.
- [26] E. W. Dijkstra, "Complexity controlled by hierarchical ordering of function and variability," in *Software Engineering*, P. Naur and B. Randell, Eds. NATO, Jan. 1969.
- [27] H. D. Mills, "Mathematical foundations for structured programming," IBM-FSD, rep. FSC 72-6912, Feb. 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*, pp. 38-54, June 1975.
- [30] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
- [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," *CACM*, pp. 1053-1058, Dec. 1972.
- [33] D. L. Parnas, "The influence of software structure on reliability," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 358-362, available from IEEE.
- [34] M. Hamilton and S. Zeldin, "Higher order software—A methodology for defining software," *IEEE Trans. Software Eng.*, pp. 9-32, Mar. 1976.
- [35] "HIPO—A design aid and documentation technique," IBM, rep. GC20-1851-0, Oct. 1974.
- [36] J. Mortison, "Tools and techniques for software development process visibility and control," in *Proc. ACM Comput. Sci. Conf.*, Feb. 1976.
- [37] I. Nassi and B. Schneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, pp. 12-26, Aug. 1973.
- [38] D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software engineering: Process, principles, and goals," *Computer*, pp. 17-27, May 1975.
- [39] M. A. Jackson, *Principles of Program Design*. New York: Academic, 1975.
- [40] P. Henderson and R. A. Snowden, "A tool for structured program development," in *Proc. 1974 IFIP Cong.*, pp. 204-207.
- [41] L. C. Carpenter and L. L. Tripp, "Software design validation tool," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 395-400, available from IEEE.
- [42] S. H. Caine and E. K. Gordon, "PDL: A tool for software design," in *Proc. 1975 Nat. Comput. Conf.*, pp. 271-276.
- [43] H. D. Mills, "Structured programming in large systems," IBM-FSD, Nov. 1970.
- [44] C. A. R. Hoare, "Notes on data structuring," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972.
- [45] N. Wirth, "An assessment of the programming language Pascal," *IEEE Trans. Software Eng.*, pp. 192-198, June 1975.
- [46] P. Brinch-Hansen, "The programming language concurrent Pascal," *IEEE Trans. Software Eng.*, pp. 199-208, June 1975.
- [47] W. A. Wolf, *ALPHARD: Toward a language to support structured programs*, Carnegie-Mellon Univ., Pittsburgh, PA, internal rep., Apr. 30, 1974.
- [48] B. H. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, pp. 50-59, April 1974.
- [49] F. T. Baker, "Structured programming in a production programming environment," *IEEE Trans. Software Eng.*, pp. 241-252, June 1975.
- [50] R. D. Williams, "Managing the development of reliable software," *Proc. 1975 Int. Conf. Reliable Software* April 1975, pp. 3-8, available from IEEE.
- [51] J. Witt, "The COLUMBUS approach," *IEEE Trans. Software*

- Eng., pp. 358-363, Dec. 1975.
- [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, NJ: Hayden, 1975.
- [54] W. A. Whitaker et al., "Department of Defense requirements for high order computer programming languages: 'Finman,'" Defense Advanced Research Projects Agency, Apr. 1976.
- [55] *Proc. 1973 IEEE Symp. Comput. Software Reliability*, Apr.-May 1973.
- [56] E. C. Nelson, "A statistical basis for software reliability assessment," TRW Systems, Redondo Beach, CA, rep. TRW-SS-73-03, Mar. 1973.
- [57] J. R. Brown and M. Lipow, "Testing for software reliability," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 518-527.
- [58] J. D. Musa, "Theory of software reliability and its application," *IEEE Trans. Software Eng.*, pp. 312-327, Sept. 1975.
- [59] R. J. Rubey, J. A. Dana, and P. W. Biche, "Quantitative Aspects of software validation," *IEEE Trans. Software Eng.*, pp. 150-155, June 1975.
- [60] T. A. Thayer, M. Lipow, and E. C. Nelson, "Software reliability study," TRW Systems, Redondo Beach, CA, rep. to RADC, Contract F30602-74-C-0036, Mar. 1976.
- [61] D. J. Reifer, "Automated aids for reliable software," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 131-142.
- [62] C. V. Ramamoorthy and S. B. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Trans. Software Eng.*, pp. 46-58, Mar. 1975.
- [63] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, pp. 156-173, June 1975.
- [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, Nov. 1975, pp. 45-88.
- [65] J. C. King, "A new approach to program testing," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 228-233.
- [66] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—A formal system for testing and debugging programs," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 234-245.
- [67] J. Goldberg, Ed., *Proc. Symp. High Cost of Software*, Stanford Research Institute, Stanford, CA, Sept. 1973, p. 63.
- [68] L. C. Ragland, "A verified program verifier," Ph.D. dissertation, Univ. of Texas, Austin, 1973.
- [69] D. I. Good, R. L. London, and W. W. Bledsoe, "An interactive program verification system," *IEEE Trans. Software Eng.*, pp. 59-67, Mar. 1975.
- [70] F. W. von Henke and D. C. Luckham, "A methodology for verifying programs," in *Proc. 1975 Int. Conf. Reliable Software*, pp. 156-164, Apr. 1975.
- [71] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 325-355, 1973.
- [72] R. L. London, "A view of program verification," in *Proc. 1975 Int. Conf. Reliable Software*, Apr. 1975, pp. 534-545.
- [73] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. New York: Academic, 1972.
- [74] W. A. Wulf, "Reliable hardware-software architectures," *IEEE Trans. Software Eng.*, pp. 233-240, June 1975.
- [75] J. Goldberg, "New problems in fault-tolerant computing," in *Proc. 1975 Int. Symp. Fault-Tolerant Computing*, Paris, France, pp. 29-36, June 1975.
- [76] B. Randell, "System structure for software fault-tolerance," *IEEE Trans. Software Eng.*, pp. 220-232, June 1975.
- [77] R. K. McClean and B. Press, "Improved techniques for reliable software using microprogrammed diagnostic emulation," in *Proc. IFAC Cong.*, Vol. IV, Aug. 1975.
- [78] E. B. Swanson, "The dimensions of maintenance," in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [79] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. IEEE/ACM 2nd Int. Conf. Software Eng.*, Oct. 1976.
- [80] J. L. Elshoff, "An analysis of some commercial PL/I programs," *IEEE Trans. Software Eng.*, pp. 113-120, June 1976.
- [81] W. L. Trainor, "Software: From Satan to saviour," in *Proc., NAECON*, May 1973.
- [82] E. H. Sibley, Ed., *ACM Comput. Surveys (Special Issue on Data Base Management Systems)*, Mar. 1976.
- [83] *Defense Management J. (Special Issue on Software Management)*, vol. II, Oct. 1975.
- [84] L. A. Belady and M. M. Lehman, "The evolution dynamics of large programs," IBM Research, Sept. 1975.
- [85] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
- [86] E. Horowitz, Ed., *Practical Strategies for Developing Large-Scale Software*. Reading, MA: Addison-Wesley, 1975.
- [87] G. F. Weinworm, Ed., *On the Management of Computer Programming*. New York: Auerbach, 1970.
- [88] P. Naur and B. Randell, Eds., *Software Engineering*, NATO, Jan. 1969.
- [89] P. J. Metzger, *Managing a Programming Project*. Englewood Cliffs, NJ: 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. Rudge and L. E. Johnson, *Effective Management of Computer Software*. Homewood, IL: Dow Jones-Irwin, 1973.
- [93] T. R. Gildersleeve, *Data Processing Project Management*. New York: Van Nostrand Reinhold, 1974.
- [94] G. F. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
- [95] J. D. Aron, *The Program Development Process: The Individual Programmer*. Reading, MA: Addison-Wesley, 1974.
- [96] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, 1974.
- [97] M. H. Halstead, "Toward a theoretical basis for estimating programming effort," in *Proc. Ass. Comput. Mach. Conf.*, Oct. 1975, pp. 222-224.
- [98] *Summary Notes, Government/Industry Software Sizing and Costing Workshop*, USAF Electron. Syst. Div., Oct. 1974.
- [99] B. S. Barry and J. J. Naughton, "Chief programmer team operations description," U. S. Air Force, rep. RADC-TR-74-300, Vol. X (of 15-volume series), pp. 1-2-1-3.
- [100] *Software Development and Configuration Management Manual*, TRW Systems, Redondo Beach, CA, rep. TRW-SS-73-07, Dec. 1973.
- [101] H. Pratman and T. Court, "The software factory," *Computer*, pp. 28-37, May 1975.
- [102] "Systems design laboratory: Preliminary design report," Naval Electronics Lab. Center, Preliminary Working Paper, TN-3145, Mar. 1976.
- [103] W. E. Carlson and S. D. Crocker, "The impact of networks on the software marketplace," in *Proc. EASCON*, Oct. 1974.
- [104] "Management of computer resources in major defense systems," Department of Defense, Directive 6000.29, Apr. 1976.



Barry W. Boehm received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957, and the M.A. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, respectively.

He entered the computing field as a Programmer in 1955 and has variously served as a Numerical Analyst, System Analyst, Information System Development Project Leader, Manager of groups performing such tasks, Head of the Information Sciences Department at the Rand Corporation, and as Director of the 1971 Air Force CCIP-85 study. He is currently Director of Software Research and Technology within the TRW Systems and Energy Group, Redondo Beach, CA. He is the author of papers on a number of computing subjects, most recently in the area of software requirements analysis and design technology. He serves on several governmental advisory committees, and current Chairman of the NASA Research and Technology Advisory Comm on Guidance, Control, and Information Systems.

Dr. Boehm is a member of the IEEE Computer Society, in which he currently serves on the Editorial Board of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and on the Technical Committee on Software Engineering.



## STRUCTURED WALK-THROUGHS: A PROJECT MANAGEMENT TOOL

IBM CORPORATION

*Abstract:* This document describes the structured walk-through, a tool being used within the IBM Systems Development Division. Experience to date indicates that there are major benefits to its use both the programming project team, and for the quality of the software they produce.

This description of structured walk-throughs represents the collected work of many people with the Systems Development Division. Special acknowledgements are expressed to William B. Cammack, David R. McRitchie, David E. Fishlock, and Henry J. Rodgers, Jr.

### 1. STRUCTURED WALK-THROUGHS

Project management has long recognized the need for periodic reviews as a vehicle for determining where the project stands in relation to its schedule, and for identifying areas that require special attention. Generally, however, these exercises have been looked upon with misgivings by those who must submit themselves to the review.

The situation which classically arises during the review is one of conflict and hostility. The review takes on the appearance of a witch hunt and the reviewer finds himself in the position of inquisitor. At best the reviewees feel they have little to gain from this encounter and most probably feel that they will come out of the review with a list of "to-dos" which will only serve to put them farther behind in their development schedules. More damaging still is their belief that the longer the list, the longer the indictment against them. They feel that they will learn nothing in the review which will help them attack their unique problems; and moreover, they feel that they will spend a large and unproductive portion of the meeting just bringing the reviewer up from ground zero.

The structured walk-through described here increases the value of these reviews beyond a determination of schedule variance and problem identification, and eliminates many of the negative aspects. Within IBM the structured walk-through is:

1. A positive motivator for the project team.
2. A learning experience for the team.
3. A tool for analyzing the functional design of a system.
4. A tool for uncovering logic errors in program design.
5. A tool for eliminating coding errors before they enter the system.
6. A framework for implementing a testing strategy in parallel with development.
7. A measure of completeness.

A structured walk-through is a generic name given to a series of reviews, each with different objectives and each occurring at different times in the application development cycle. The basic characteristics of the walk-through are:

1. It is arranged and scheduled by the developer (reviewee) of the work product being reviewed.
2. Management does not attend the walk-through and it is not used as a basis for employee evaluation.

3. The participants (reviewers) are given the review materials prior to the walk-through and are expected to be familiar with them.
4. The walk-through is structured in the sense that all attendees know what is to be accomplished and what role they are to play.
5. The emphasis is on error detection rather than error correction.
6. All technical members of the project team, from most senior to most junior, have their work product reviewed.

## 2. MECHANICS

The objectives of the structured walk-through will be different at different stages of the project. The basic mechanics will, however, remain the same. The reviewee, the person whose work product is being reviewed, is responsible for arranging the meeting. Several days prior to the meeting the reviewee selects the attendees he feels are required, distributes his work product to them, states what the objectives of the walk-through will be, and specifies what roles the reviewers are to play.

Although there are no hard and fast rules as to who the reviewers should be, the idea is for the reviewee to pick those interested parties who can detect deviations, inconsistencies, and violations within the work product or in the way that it interacts with its environment. Typically, but not necessarily, the reviewers will be project teammates of the reviewee. For example, early in the project, when a major objective is to ensure that the system is functionally complete, the reviewee might want user representatives. Or, if programmers and analysts are functionally separated, and the objective of the walk-through is to ensure that the programmer's internal specifications match the analyst's external specifications, then the programmer would want the analyst to attend. Within IBM, it is not uncommon for a programmer to reschedule a walk-through several times in order to ensure that a particular reviewer will be available.

A typical walk-through will include four to six people and will last for a pre-specified time, usually one or two hours. If at the end of that time the objectives have not been met, another walk-through is scheduled for the next convenient time. Someone is designated as the recording secretary. This person records all the errors, discrepancies, exposures and inconsistencies that are uncovered during the walk-through. This record becomes an action list for the reviewee and a communication vehicle with the reviewers.

In addition to the substantive questions which will hopefully arise in the reviewer's mind prior to the walk-through, he will undoubtedly detect minor mistakes such as typos, spelling, grammatical and coding syntax errors. These can be handled several ways. One way is to instruct each reviewer to make an error list and pass it to the recording secretary at the beginning of the walk-through. Another way is for each reviewer to cover these errors with the reviewee offline. Or, the reviewers can annotate their copies of the work product and return it to the reviewer at the end of the walk-through. The important point is that the walk-through should be concerned with problems of greater substance (i.e., ambiguous specifications, basic design flaws, poor logic, inappropriate or inefficient coding techniques).

Mechanically, what takes place during the structured walk-through? First, the reviewers are requested to comment on the completeness, accuracy and general quality of the work product. Major concerns are expressed and identified as areas for potential follow-up. The reviewee then gives a brief tutorial overview of the work product. He next "walks" the reviewers through the work product in a step-by-step fashion which simulates the function under investigation. He attempts to take the reviewers through the material in enough detail so that the major concerns which were expressed earlier in the meeting will either be explained away, or brought into focus. New thoughts and concerns will arise during this "manual execution" of the function, and the ensuing discussion of these points will crystallize everyone's thinking. Significant factors that require action are recorded as they emerge.

A key element regarding the structured walk-through is its relationship to the project test strategy. Within IBM, the structured walk-through is part and parcel of a parallel test strategy, and in fact, the "manual execution" is often driven by formalized test cases. This is discussed more fully in Section 4.

Immediately after the meeting, the recording secretary distributes copies of the handwritten action list to all the attendees. It is the responsibility of the reviewee to ensure that the points of concern on the

action list are successfully resolved, and that the reviewers are notified of the actions and/or corrections that have been taken. (This latter point is important because many of the revelations which arise impact the reviewers, particularly if they and the reviewee are teammates.) Management does not double check the action list to ensure that the outstanding problems have been resolved, nor does it use this list as a basis for employee evaluation. Rather the action list is considered to be a tool used to improve the product.

35

### 3. AS PART OF NEW TECHNOLOGIES . . .

Structured walk-throughs have been implemented within IBM programming groups which are using structured programming, top-down development, development support libraries and team operations. In fact, the use of walk-through as described in this paper has evolved to its present position because of these new technologies.

The visibility inherent in structured programming, the idea that code is meant to be read by others, the enforced programming conventions, and the simplified program logic make it easy for the reviewer to be "walked through" code segments.

The use of HIPO as a top-down design and documentation tool lends itself well to the structured walk-through. HIPO's graphical representation of function gives the reviewee the luxury of something concrete and tangible through which he can take the reviewers in a step-by-step fashion at increasing levels of detail.

A development support library organizes and structures the emerging system so that the details can be easily reviewed. In addition, the librarian can also serve as the recording secretary for the walk-throughs.

The concept of a tightly knit team whose members possess unique skills and who are in close communication with each other, is logically supported by the idea of a walk-through. Since the chief programmer and the backup programmer already read code, the extension to everyone reading code is not a major jump. Additionally, there is value in the walk-through as an educational tool. Because the chief programmer and the backup programmer design and code the top of the system first, their initial walk-throughs serve as important learning experiences for the other team members-- both in terms of design and coding techniques, and as an introduction to the system.

Within an application development cycle, there are several major milestones and many minor milestones where the walk-through technique can be used. As an example, a manning curve for an application development cycle in which the new technologies are being used might look as shown in Figure 1. The management of this project could decide that one condition for successfully reaching the milestones listed in the left hand column of Figure 2, is that the items in the right hand column must have been reviewed in a structured walk-through. In this sense the walk-through tracks progress and serves as a meaningful measure of completeness. Major milestones where structured walk-throughs might be employed include end of system planning, end of system design and end of development.

### 4. PARALLEL TESTING

The structured walk-through can serve to establish a framework for parallel testing. Parallel testing implies: 1) the development of test cases and testing procedures in parallel with the development of the system, and 2) an independent tester who is responsible for implementing the test strategy.

When using team operations, the tester would logically be the backup programmer. In large, functionally separated, organizations the tester(s) might come from an independent group.

The tester builds a product in much the same manner as the developer does. They both start at the same place, with a set of functional specifications. The developer, however, looks at the specs as a builder might look at blueprints, while the tester looks at those specs in the way a building inspector might look at blueprints. The tester, like the inspector, attempts to ensure that the specifications meet certain standards, and that the product matches the specifications.

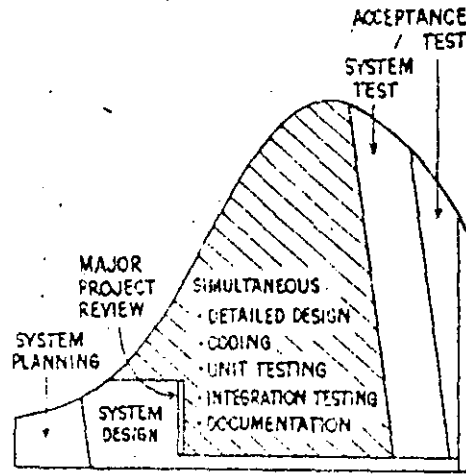


Figure 1. A typical manning curve for an application development cycle.

A functional program specification can be boiled down to a set of cause and effect relationships:

- "If the accumulated FICA deduction is equal to or greater than \$10,800, then return the difference to net pay."
- "When the on-hand balance falls below the reorder point, transfer control to the EOQ routine."
- "Set the transmission line to inoperative and notify the network control operator if the retry procedure fails."

Initially the tester takes the functional specifications and breaks them down into a series of cause and effect statements. Rigorous testing means that each of these cause and effect relationships must be tested. That is to say, the tester, using some form of tabular or graphical assistance, must determine whether each cause has its desired effect. Unfortunately, this is not always easy to do. If it were, testing would not be a problem and systems would be more error free. Cause and effect relationships tend to string together in complex logical chains. Therefore, it is not always obvious what is a cause and what is an effect. In addition, analysts and designers don't apply the same discipline to their specifications that the programmer must apply to his code. Rather, they tend toward free flowing prose, resplendent with inconsistencies.\* Nevertheless, the product which the tester is creating will evolve into a formalized set of machine readable test cases, residing in a test library, which based on the quality of his efforts and the thoroughness with which he breaks down the functional specifications, will test the code.

Within IBM the tester plays a key role in those structured walk-throughs which relate to detailed design and programming. The tester views the walk-through as the vehicle which formally brings him together with the developer. After the reviewee walks the reviewers through the work product to bring everyone to a common level of understanding, he passes control of the meeting to the tester. The tester presents his test cases, one by one, to the reviewee. All participants observe as the reviewee walks each test case through the work product. Inconsistencies and errors are spotted in the work product

\* The English language is not noted for its ability to express complex relationships with precision. Perhaps the future will see us evolve into structured specification languages. A step in that direction would be pseudo code narrative associated with structured programming.

PROJECT MILESTONES	ITEMS TO BE REVIEWED VIA A STRUCTURED WALK-THROUGH
End of System Planning	Project Plans System Definition Task Identification
Major Project Milestones	
Major Project Review "Technical"	Functional Specifications Work Assignments Schedules
Multiple Minor Milestones	
• Detailed Design	Internal Specifications HIPO Package
• Doding	Uncompiled Source Listings
• Documentation	User Guides Programmer Maintenance Manuals • Internal Specifications • HIPO Package
End of Development	Deliverable Product • Code • Documentation

Figure 2. The table shows items which might be reviewed using structured walk-throughs at various times during a project. The minor milestones would be repeated as the system grew.

and also in the test cases. The recording secretary is responsible for recording problems that relate to the product, and the tester is responsible for recording and correcting problems that relate to his test cases. The tester's goal is to produce a complete and non-overlapping library of test cases which will validate the final product.

The evolution of the test library proceeds in parallel with the system. While the system develops from functional specifications to internal program specifications and HIPO diagrams, to source code and finally to compiled code, the tester is independently developing the test library from the functional specifications, to cause and effect relationships, to manual test cases, and finally to machine readable test cases. By the time a subset of the system is ready to be compiled, the test cases will be included in the test library and can be driven against the compiled code.

This parallel evolution of the application and its test cases, synchronized at each development step by a structured walk-through, ensures a thoroughness and a discipline which cannot be achieved when testing is handled as a follow-on to development.

### 5. PSYCHOLOGY

The interested reader may wonder why management doesn't take a more active part in the walk-through; or more specifically, why management doesn't use the action list as a measure of employee performance. The answer is that management could, but only at the expense of losing some of the values of the walk-through.

An essential ingredient for a successful walk-through is an open and non-defensive attitude on the part of the participants. A productive atmosphere is one in which the reviewee makes it easy for the reviewers to find problems. He should welcome their feedback and should encourage their frankness. If, however, he feels that he is being evaluated by what occurs in the walk-through, and by the size of the action list, he will naturally tend to suppress criticisms. He will be defensive and unreceptive to new ideas. His ego will be staked to the work product and he will have little motivation to use the session as a learning experience. A successful walk-through, by comparison, is one in which many errors and inconsistencies are uncovered.

The role of the reviewers is one of preparation, non-malicious probing, and problem definition. If they are teammates of the reviewee, it will not be uncommon for them to discover that hidden relationships exist between what they are developing and what is being reviewed. Ambiguities will come to light which will require further clarification and definition. If for no other reason, management should value the walk-through for its contribution as a communication tool among the developers.

Setting the proper psychological atmosphere for structured walk-through is key. An organization utilizing team operations, top-down development, and structured programming can do it rather naturally. Since the chief programmer and the backup programmer will produce the initial design and the most critical code in the system, their work products will be the first under review. Because they are more senior and more closely attuned to management's desires (the design programmer may in fact be the manager), they are in a position to establish the proper framework and attitude surrounding the walk-through. In addition, these initial walk-throughs will serve as a learning experience for the team not only as to the walk-through mechanics, but with respect to the system itself.

## 6. SUMMARY

Our experience with structured walk-throughs has been most encouraging. Undoubtedly there are a number of ways they could be modified to fit other organizations. The central idea, however, should remain the same: i.e., to convert the classical project review into a productive working session which not only tracks progress but which makes a positive contribution to that progress. Outwardly management involvement appears low, but in reality structured walk-throughs provide management with a vehicle for catching errors in the system at the earliest possible time when the cost of correcting them is lowest and their impact is smallest. ■



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

CAPITAL-INTENSIVE SOFTWARE TECHNOLOGY

DICIEMBRE, 1984





# Capital-Intensive Software Technology

Peter Wegner  
Brown University

Each section of this four-part article deals with a different aspect of capital-intensive software technology. Together, they present an integrated view of the subject.

## Introduction

### Capital-intensive technology and reusability

What is capital?

Capital is a stock rather than a flow. In its broadest sense it includes the human population; non-material elements such as skills, abilities, and education; land, buildings, equipment of all kinds; and all stocks of goods, finished or unfinished, in the hands of both firms and households.

—*Encyclopaedia Britannica*, 1968

To flirt is capital.

—*The Mikado*, Gilbert and Sullivan

Striking similarities between industrial and software technology have led to considerable borrowing of the terminology of industrial technology for corresponding concepts of software technology. For example, the term "software engineering" emphasizes that the construction of soft-

ware is an engineering task. Terms such as "software tools" and "software factory" suggest that paradigms of industrial production are being adopted for software production.

The terms "capital" and "capital-intensive," first introduced in the context of industrial technology, can be

## Electronic steam engines

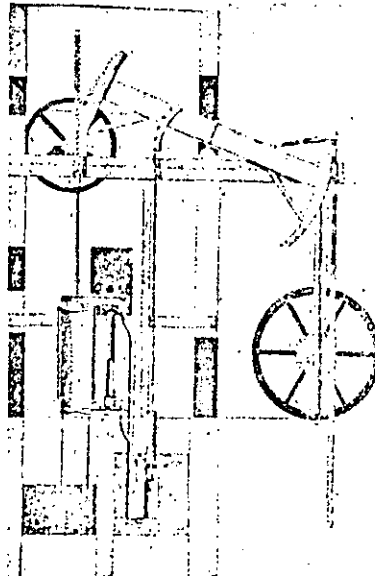
2

Large central processors are the steam engines of the computer revolution. The shift from large central processors to personal computers is comparable to the shift from steam engines to combustion engines and electric motors. The nineteenth century transition from cumbersome energy supplies to cheaper, more accessible sources of energy is being paralleled in the 1980's by a shift from inaccessible large computing engines to accessible small computing engines in every home and appliance.

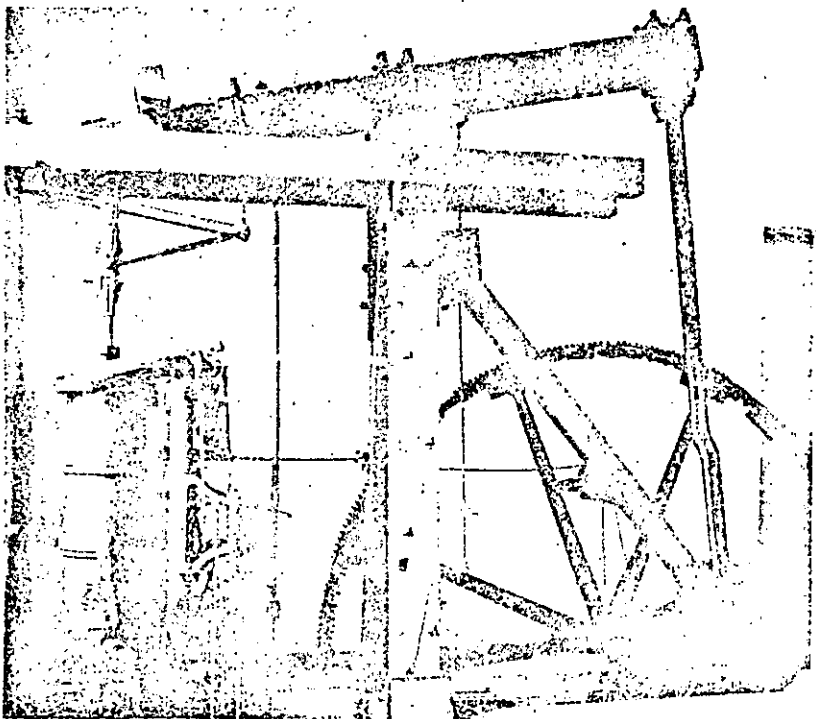
Watt's engine, shown at right and below, marked the real beginning of the age of steam. In 1765, while repairing a Newcomen pump, James Watt recognized one of the machine's main disadvantages. Condensation of the steam inside the cylinder lowered the pressure required for suction. Newcomen's machine introduced cold water to condense the vapor, but every time the steam condensed, the cylinder cooled off. Thus, much of the new steam was wasted in reheating the cylinder.

Watt's first innovation was the condenser, a separate compartment in which the steam was made to condense. He kept the cylinder insulated by surrounding it with a steam jacket. His most important contribution, however, was in obtaining rotary motion. In 1782, he constructed a double-acting engine that, through a series of cogged wheels, transformed the rocker arm's alternating movement into a rotary movement. Later, Watt equipped his engine with a governor and a pressure gauge.

The first Watt engine was installed in a coal mine in 1784, the same year that Arkwright and Crompton achieved the complete mechanization of spinning. The industrial revolution was entering its most active phase, and within a few decades, the technological picture changed radically. By 1800, 52 of Watt's engines were operating in various types of mines, one had been applied to a drop hammer, and 84 had been installed in cotton mills.



The Beilmann Archive



The Beilmann Archive

applied to software technology. Software technology, like the technology that fueled the industrial revolution, was labor-intensive in its youth and is becoming capital-intensive as it matures.

Economists such as Adam Smith used the term "capital," along with "land" and "labor," as one of three factors of production. Because Karl Marx, in his book *Das Kapital*, emphasized the exploitation resulting from the ruthless use of capital to maximize profits, the term came to have bad connotations. The ensuing arguments between classical and Marxist economists about who should own capital resources have sometimes obscured the more central question of how capital resources should be harnessed for the benefit of mankind.

We are here concerned with the public benefits of capital, irrespective of ownership. Our purpose is to understand how capital goods enhance our productivity in building bigger and better software systems and, more generally, in managing our growing stock of knowledge. Intuition suggests that a production process is capital-intensive if it requires expensive tools or if it involves large startup expenditures. Software development is becoming increasingly capital-intensive: its tools are becoming more powerful and expensive, and it requires greater early investment to reduce later expenditures.

Machine tools of the industrial revolution and software tools such as compilers are both reusable resources. Moreover, any reusable resource may be thought of as a capital good whose development cost may be recovered over its set of uses. Thus, it seems reasonable to identify the notion of capital goods with that of reusable resources and the notion of capital with that of reusability.

The idea of reusability subsumes the economic notion of capital but is more general. It includes capital resources not only for industrial technology but also for software technology, research, and education. While economists reserve the term "capital" for reusable industrial resources, the no-

tion of reusability is domain independent. The characterization of capital in terms of reusability is not just terminology; it is a technical sleight of hand; it provides some real insight into mechanisms for enhancing productivity and reliability in any technology.

Capital goods such as a lathe or an assembly line are reusable resources for producing consumer goods. Capital goods such as compilers and operating systems are reusable resources for producing application programs. Programmers are reusable resources in the production of programs. Activities such as education which contribute to programmer productivity are capital-intensive in that they enhance the reusability of people.

Technologies that rely heavily on capital goods are called capital-intensive technologies. The process of developing capital goods is called capital formation. Capital formation in software technology is dependent on the implementation of concepts and models rather than on the construction of physical machines. Our generalized notion of capital includes both conceptual and physical capital formation because we see reusability as a key denominator.

Reusability is a general engineering principle whose importance derives from the desire to avoid duplication and to capture commonality in undertaking classes of inherently similar tasks. It provides both an intellectual justification for research that simplifies and unifies classes of phenomena and an economic justification for developing reusable software products that make computers and programmers more productive. The assertion that we should stand on each other's shoulders rather than on each other's feet may be interpreted as a plea for both intellectual and economic reusability.

The initial economic motivation for the development of general-purpose computers was the reusability of computer hardware. General-purpose computers are a capital-intensive response to the information processing needs of society that allow critical computing resources such as the cen-

tral processing unit to be reused one million times per second. Less critical resources such as the computer memory may be reused for programs and data with very different behavioral characteristics.

The changed economic balance between hardware and software has resulted in changed perceptions of what is capital-intensive. When hardware was the dominant cost in a computer system, attention focused on computer efficiency. Even Fortran was regarded with skepticism because its compiled code might be less efficient than machine language. Time-sharing operating systems were carefully crafted so that a single powerful processor could be shared (reused) by many users. With decreasing hardware costs, attention has shifted from the reusability of central processing units to the reusability of software and the productive use of people.

Technological changes which took several decades in the industrial revolution are being compressed in the computer revolution into a much shorter time. The greater speed of technical change means that capital investment must be recovered more quickly and that enhancement and evolution consume proportionately more resources than in a slowly changing technology. This contributes to the fact that maintenance and enhancement are the dominant costs in the software life cycle today.

## Overview and organization

The drive to create reusable rather than transitory artifacts has aesthetic and intellectual as well as economic motivations and is part of man's desire for immortality. It distinguishes man from other creatures and civilized from primitive societies.

We explore a variety of capital-intensive software activities, including (1) software components, (2) programming in the large, (3) knowledge engineering, and (4) accomplishments and deficiencies of Ada. Each topic is presented as a self-contained section that can be read independently. However, the article as a whole presents an integrated view of capital-intensive

software technology that is greater than the sum of its parts.

**Part 1.** Software components are the capital-intensive building blocks out of which large programs are constructed. We review the evolution of software components and examine the relation among subprograms, data and process abstraction, and object-oriented programming. We contrast the computation model of block-structured languages with that of distributed programming languages. The role of libraries as repositories of knowledge that organize the interaction of software components during both program development and program execution is examined. A taxonomy is developed which suggests that the study of software components is maturing into a subdiscipline of computer science with considerable structure and substance.

**Part 2.** The evolution of life-cycle models from the waterfall model through the operational (rapid prototyping) model to the knowledge-based model is examined. The tension between efficiency and modifiability in the design of large systems is discussed. We give examples of the reusability of concepts in both theoretical and experimental computer science, indicating that the value of research contributions and concepts can be measured by the same metric as software products. Application generators generate software components of high granularity in a specialized domain and take advantage of the reusability of the generating mechanism and of the reusability of the environment in which generated software components are embedded.

**Part 3.** Software technology is concerned not only with amplifying the productivity of the programmer but also with amplifying man's mental capacities in other areas. We suggest that knowledge engineering will play the same role in the management of knowledge that software engineering plays in the management of software.

The syntactic interface of a subprogram definition may be viewed as a socket, and subprogram calls may be viewed as plugs that are plugged in at the time of subprogram call. Subprogram parameters may be viewed as prongs whose size and shape depend on the parameter type. The number and type of prongs of a subprogram call must match the number and type of corresponding slots in the socket corresponding to the subprogram definition (see Figure 1).

Syntactic interface specifications are a weak form of specification sufficient to determine that components fit together correctly, but insufficient to determine the correctness of computations of the resulting software structure. However, weak interface specifications are tractable and useful in the sense that they allow consistency between specifications and invocations of software components to be checked and enforced at compile time. Strong semantic interface specifications are

intractable in the sense that they do not always exist and their correctness cannot always be verified. This distinction is intrinsic in the language design of Ada.

Ada separates the syntactic interface specification of a component from the body that implements the component. Ada's weak syntactic interface specifications have been criticized on the grounds that they are insufficient to determine program correctness. On the other hand, it can be argued that the design decision to use weak interface specifications as a basis for enforcing compile-time interface consistency is in fact one of Ada's strengths and represents an important contribution to language design.

One of the purposes of an interface specification is to capture invariant properties of the static program so that they can be checked and enforced by the compiler. Strongly-typed languages such as Ada enforce compile-time type consistency by interface specifications. The programming language NIL,\* developed by IBM at the T. J. Watson Research Center in Yorktown Heights, is designed so that not only the type but also the state of initialization (type state) of variables is a compile-time invariant of the static program.<sup>2</sup> It permits stronger compile-time invariants to be specified in the interface, stronger compile-time consistency checks that guarantee proper initialization of variables before they are used, and finalization of variables after they are used.

Programming systems may be characterized in terms of the internal interface specifications among their components. Interlisp and Unix have weak interface specifications, while languages for application programming have strong interface specifications. The Ada environment requirements were a pioneering attempt to specify programming systems out of strongly typed components.

\*The acronym NIL stands for Network Implementation Language. It has nothing to do with the programming language Lisp.

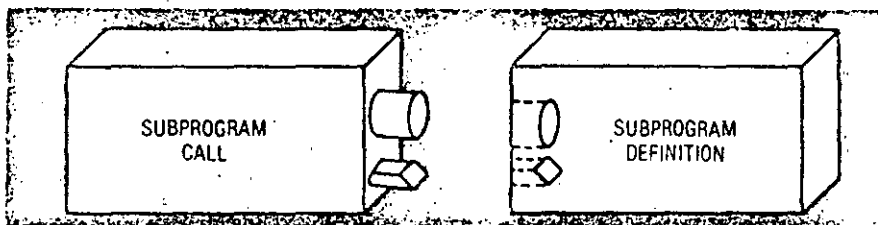


Figure 1. Plug-and-socket model for subprograms.

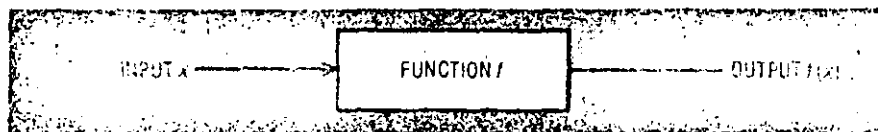


Figure 2. Function abstraction.

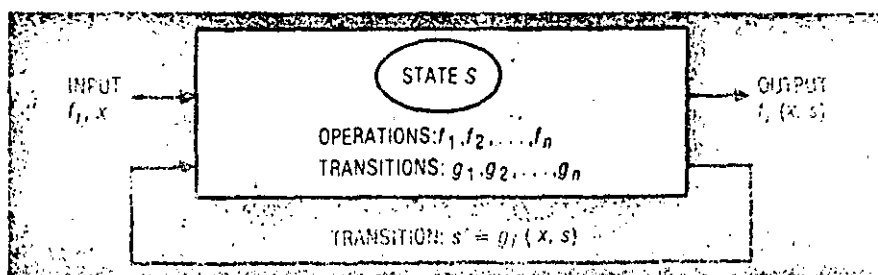


Figure 3. Data abstraction.

NIL's notion of interfaces is even stronger than that of Ada since not only the type but also the state of initialization of variables is known at compile time.

The trade-offs between the flexibility and efficiency of very weak interface specifications and the guaranteed integrity of strong interface specifications need to be better understood. Strong interfaces serve to increase productivity during program development but may limit expressive power. Their cost-effectiveness is greater for application programs, where development and maintenance are the primary bottleneck, than for system programs, where efficiency is the primary consideration. In principle, a good compiler should be able to transform strongly typed modules specified by the user into efficient untyped internal modules, but we do not yet have sufficient experience to do this well.

### Function and data abstraction

In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences.

—C. A. R. Hoare,  
*Notes on Data Structuring*, 1972

Many different abstraction mechanisms have been proposed as the basis for a software components industry, each representing different building blocks from which programs can be constructed and each resulting in different paradigms (methodologies) for programming. In this section we describe the features of function and data abstraction, and illustrate design and interface issues with examples from programming languages such as Ada.

Function abstractions may be specified by input-output relations in which every input  $x$  determines a unique output  $f(x)$  that depends only on the input  $x$  and on no other data

(see Figure 2). The user is aware only of the input-output specification and not of the way the function is implemented. The specification constitutes the interface to the user, and the implementation is hidden from the user.

Function abstraction may be contrasted with data abstraction in which the information hidden from the user includes data as well as function implementations. Data abstractions have an internal state that "remembers" the effect of past operations and allows components to use past experience to modify future behavior. The effect of an operation  $f$  on an input  $x$  is no longer uniquely determined but may depend on the internal state  $s$  (see Figure 3). An operation  $f$  on a state  $x$  may result in an output,  $y = f(x, s)$ , and in a new state,  $s' = g(x, s)$ , just as in finite automata. A given data abstraction may in general support more than one operation that shares the common data structure. For example, a data abstraction for a stack generally supports push and pop operations and a test for the empty stack, all of which operate on the shared common state.

Whereas subprogram interfaces specify a single function  $f$ , data abstractions may specify a set of operations  $f_1, f_2, \dots, f_N$ , each associated with a hidden state transition function  $g_1, g_2, \dots, g_N$ . Each operation may be viewed as a socket into which users who call the data abstraction are plugged for the duration of a call and are then unplugged.

In designing interfaces for data abstraction, the following issues must be addressed:

(1) What kinds of resources should data abstractions provide to their users? Should they provide operations, types, variables, or some subset of these resources?

(2) What rules should govern the granting of access rights to users of an abstraction?

Ada allows package interfaces to contain operations, types, variables, and other linguistic constructs. This wide interface (you can drive a truck through it) may be contrasted with

the narrow interfaces of CLU, which permits just operations to be specified in the interface, and of NIL, which permits just types (of messages) to be specified in the interface.

Updatable interfaces may increase accessing efficiency but cause the component to lose control over information in the interface. Moreover, they violate the principle that interfaces be compile-time invariants. Interfaces become dependent on values of variables rather than on invariant interconnection properties of components. The trade-offs between efficiency and integrity in choosing between wide and narrow interfaces are similar to those for global variables.

In Ada, access rights to abstractions declared in an enclosing block are inherited through the block structure mechanism. Access rights to library components are not restricted but must be redundantly mentioned in "with" clauses, thereby allowing the compiler to track dependencies among components and facilitating compile-time type checking for imported resources. Thus, "with  $Q$ " placed before a component  $P$  specifies that the resources of  $Q$  are imported into  $P$ .

Ada's compile-time binding of component interdependence may be contrasted to runtime interconnection facilities provided by operating systems. NIL embeds operating system facilities in a strongly typed programming language. It allows the programmer to establish dynamic interconnections by treating ports as updatable variables whose values are connections to ports in other processes. Here again, there are trade-offs between the efficiency of compile-time binding and the flexibility of dynamic network interconnections.

Function and data abstraction determine different paradigms for programming associated with different partitionings of a computation into reusable and varying parts. Function abstraction emphasizes reusability of functions for varying data, while data abstraction emphasizes the reusability of data objects for various operations that may be applied to them. Function abstraction is based on a paradigm in

which programs are the primary capital goods and data are considered to be consumer goods, supplied as input by the consumer and returned as output to the consumer. The data abstraction paradigm views data objects as the primary reusable resource and sees functions as consumers with a shorter lifetime than the objects on which they operate.

## Process abstraction

**Functions are abstract operations. Data abstractions are abstract variables. Processes are abstract computers.**

Process abstractions are similar to data abstractions in having an internal state and a collection of operations that may transform the internal state. They differ from data abstractions in having an independently executing thread of control that determines the order in which operations become available for execution. They have ports through which users may obtain synchronized access to resources of the process. Access requests are placed in a queue from which they are removed only when the process is ready to handle them (see Figure 4).

Two kinds of processes may be distinguished:

(1) *concurrent processes*, which

may communicate through shared data in a global memory, and **10**

(2) *distributed processes* with no shared data that communicate only by message passing.

Concurrent processes require a mechanism for protecting shared data from concurrent access. This may be accomplished by monitors,<sup>3</sup> which protect the set of operations of a data abstraction from concurrent access by queuing all calls in a sequentially executed monitor queue. An alternative mechanism is the atomic objects of Argus,<sup>4</sup> which permit concurrent read operations on a data abstraction but protect against concurrency during write operations.

Distributed processes do not need a mechanism for protecting data from concurrent access since there is no data outside a distributed process that needs to be protected. Ports can serve as the mechanism for data protection as well as the mechanism for process synchronization, thereby achieving linguistic economy.

In designing interfaces for process abstraction the following issues must be addressed:

(1) Are there essential differences between the interface needs of data and process abstraction?

(2) What relation between interfaces and components is needed to support

program evolution during both the development and the execution of long-lived distributed embedded systems?

The differences between the interface properties of Ada's data and process abstractions appear to be somewhat arbitrary. Ada's process (task) interfaces are narrower than its data abstraction interfaces, containing only entry points (operations). Process abstractions are first-class objects, which may be passed as parameters and appear as components of records and arrays, while sub-program and data abstractions are second-class objects.<sup>5</sup>

Ada allows interface specifications for both data and process abstraction to be specified and compiled independently of their body (implementation). This represents an important step forward in language design because it facilitates the use of software components in building large programs. Interfaces are specified and compiled early in program development since the resources they define may be needed by other components. Bodies that implement interface specifications are programmed much later, by "body shops," since other components do not care how bodies are implemented, provided they deliver the resources promised in their specification.

### Abstraction and reusability

The relation between function and data abstraction is captured in lambda notation by the expressions  $[\lambda x.f(x)]$  and  $[\lambda f.f(x)]$ . The function abstraction  $[\lambda x.f(x)]$  can be applied to an argument  $a$  in the domain of  $f$  to yield  $f(a)$ . It captures the reusability of the function  $f$  for a range of values of the variable  $x$ . In contrast the expression  $[\lambda f.f(x)]$  allows the set of functions applicable to  $x$  to vary. It may be thought of as a data abstraction because it captures the reusability of a data object  $x$  for a range of applicable functions  $f$ . This example illustrates that different abstractions of the expression  $f(x)$  correspond to different choices of what is to remain fixed (reusable) and what is to be allowed to vary.

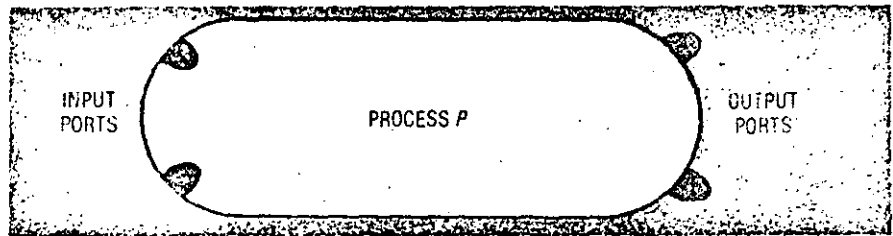


Figure 4. Process abstraction.

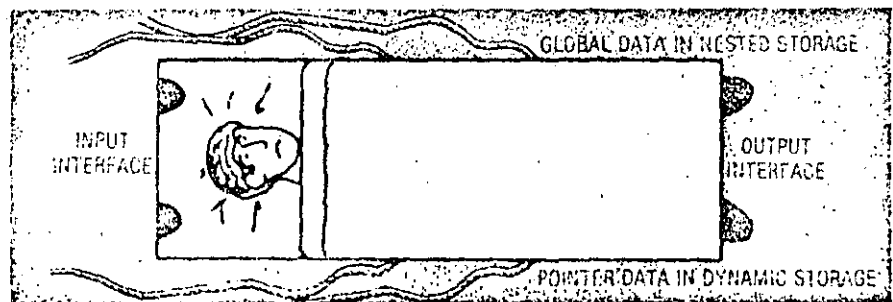


Figure 5. Global and pointer variables.

In  
of a  
e  
deve  
duri  
that  
and  
gran  
prog  
men  
ing  
seve  
utec  
vide  
whi  
N  
spe  
froi  
cifi  
cha  
tha  
a c  
not  
img  
use  
mu  
C  
ass  
use  
pu  
dy  
po  
sp  
tic  
fu  
  
se  
tic  
to  
cc  
th  
ne  
w  
er  
  
sp  
ac  
p  
st  
e  
a  
f  
N  
a  
s  
c  
t

Interface specifications and bodies of a given Ada software component are weakly coupled during program development but strongly coupled during program execution in the sense that the linkage between specifications and bodies cannot change during program execution. Ada is designed for program evolution during development, but it precludes evolution during program execution. This can be a severe limitation in long-lived, distributed embedded systems that must provide for evolution and modification while they are being used.

NIL has an approach to interface specification that is very different from that of Ada. Its interfaces specify properties of communication channels between components rather than of sockets at the receiving end of a communication channel. They are not tied to specific bodies but may be imported by any process that needs to use the interface for purposes of communication. Both calling and called processes must import the interface associated with the channel they will use to communicate. NIL permits output ports of a calling process to be dynamically connected to a new input port with a compatible interface specification during program execution. (Dynamic linking is discussed further on p. 21.)

A given NIL process may import several different interface specifications and present different interfaces to each of the processes with which it communicates. This corresponds to the intuitive notion that any component (or person) is used in different ways by different components of its environment.

While Ada's separation of interface specifications and bodies represents an advance over previous software component technology, it places constraints on execution-time program evolution that can be avoided only by an even looser binding between interfaces and bodies, such as that found in NIL. This is just one of many examples illustrating that Ada's considerable contributions to language design represent the beginning rather than the end of our quest for a stan-

dard software components technology for evolutionary, embedded computing systems.

11

## Software components of Ada

*Ada superimposes data and process abstraction on a Pascal-based language core.*

Ada supports a variety of different kinds of software components. It has subprograms for function abstraction, packages for data abstraction, and tasks for process abstraction. It is a good language to illustrate the interaction among software components, both because it supports many kinds of components and because it is wrong in interesting ways.

(1) Ada's concern with efficiency resulted in software components that can communicate not only through interfaces, but also through shared global variables declared in textually enclosing environments and through pointers to shared data in a "heap."

(2) Ada does not properly integrate its data and process abstraction mechanisms. In particular, data abstractions (packages) are not protected against concurrent access by process abstractions (tasks).

Software components should normally communicate with their clients only through their interfaces. Communication through shared global data or pointers is not properly documented in the interface and results in imperfect, unverifiable abstractions. Components with shared global data are more like patients in a hospital connected to a life-support system by a variety of tubes than like truly autonomous entities (see Figure 5).

Taking this analogy further, patients can control ingestion of substances through explicit interfaces like the mouth but have no control over substances entering the body through the life-support system. Software components in Ada can similarly control ingestion and manipulation of data through interfaces but have no control over global and pointer values.

The lack of protection of data abstraction against concurrent access by process abstractions can result in erroneous programs; with errors that cannot be caught at compile time or runtime and with unpredictable effects that may include the corruption of provably correct components. Erroneous programs violate basic modularity prerequisites, since modules that have been proved correct may be corrupted by unpredictable errors in erroneous modules.

The protection of shared data abstraction against concurrent access may be realized by (1) replacing packages by protected data abstractions such as monitors or (2) eliminating the possibility of sharing, both at the level of variables and at the level of data abstraction. Linguistic constructs that eliminate sharing are discussed in greater detail in the next section.

## Distributed processes

*The strong modularity of distributed processes has its physical origins in hardware requirements of distributed systems but derives its logical importance as a paradigm for programming in the large.*

Distributed processes model autonomous concurrently executing computers. They are both logically and linguistically simpler than concurrent processes with shared data. They provide a paradigm for a software components technology for large, long-lived, evolving programs that is more powerful than that of data abstraction. We shall briefly examine some open design issues for distributed processes.

One important design issue is whether to allow the programmer to specify internal concurrency within distributed processes. Internal concurrency models large computers with multiple processes sharing a common memory. It greatly enhances the efficiency of certain kinds of computation. But it also increases the complexity of the programming language, requiring concurrency control within distributed processes to ensure disciplined access to shared

data. Two kinds of processes are required, one to model distributed concurrency and the other to model internal concurrency within distributed processes.

Distributed processes with internal concurrency will be called *distributed concurrent processes* and may be contrasted with *distributed sequential processes* that have no internal concurrency. Distributed sequential processes have a simpler model of computation because there is only one kind of concurrency rather than two. The problem of shared local data within distributed processes may be eliminated. A single interprocess synchronization mechanism may be used for both process communication and synchronized data access.

Distributed sequential processes cannot express concurrent reading or writing of shared data structures. Access to a multi-user database such as that of an airline reservation system must be handled through a database server process that accepts queries sequentially through input ports. However, concurrency for such sequential queries may be reintroduced by an optimizing compiler.<sup>2</sup> Figure 6 illustrates how queries arriving at an input port of a distributed sequential process may be compiled into concurrent queries of a distributed concurrent process. Such a compiler maps distributed sequential processes of a high-level user interface into distributed concurrent processes of an internal language that supports disciplined concurrent access to a shared database.

Distributed sequential processes allow the user to think concurrently at the logical level using distributed processes as the unit of concurrency. But

they discourage the user from introducing explicit concurrency purely for purposes of performance. In particular, they prevent the user from specifying optimizations requiring shared concurrent access to data structures, leaving such optimizations to smart compilers.

Distributed sequential processes determine a level of logical concurrency that may be either less than or greater than the level of physical concurrency during program execution. The optimization above introduces extra concurrency at execution time to improve efficiency. The reverse situation when logical concurrency is greater than physical concurrency arises when logically distributed processes are executed at a single physical location, either concurrently or sequentially. In this case, channels between distributed processes may be represented by shared variables so that remote procedure calls may be implemented as efficient transfers of control between components that share common storage. Sharing introduced by this kind of optimization is safe because it is introduced by the system rather than the user.

Distributed sequential processes permit neither local nor global sharing of data structures, but sharing plays a key role in both optimizations above. Increased concurrency is realized by sharing of data structures local to a process, while decreased concurrency allows channels to be replaced by shared data structures global to processes. The relation between sharing and optimizations that change the level of concurrency deserves to be further explored.

Another important design decision for distributed processes is the philosophy for recovery from failures. Programmable mechanisms provide control over recovery by the user, while transparent mechanisms free the user from this responsibility and place greater responsibility on the system. Argus provides programmable mechanisms while NIL hides the recovery mechanism from the user.<sup>6</sup>

Argus programs explicitly distinguish between volatile storage, which

### Argus and NIL

Argus<sup>4</sup> supports distributed concurrent processes called guardians. Guardians communicate with each other by message passing. Concurrent processes within a guardian communicate through shared variables. Synchronization of access to shared local data is realized by atomic objects that have their own locks for read and write access control. Argus has two synchronization mechanisms: remote procedure calls for inter-process synchronization between guardians, and atomic objects for shared data synchronization within guardians.

NIL<sup>2</sup> supports distributed sequential processes. Its primitives are simpler than those of Argus because there is just one kind of concurrency and no shared variables. It does not allow concurrent access to shared data to be specified at the user level. Providing concurrency for logically sequential queries of a database is regarded in NIL as an optimization that is the responsibility of the system rather than the user. If efficient and reliable mechanisms for realizing such concurrency can be developed, possibly with the aid of pragmas that allow the user to indicate input ports for which concurrency optimizations are appropriate, then the NIL approach will become state of the art.

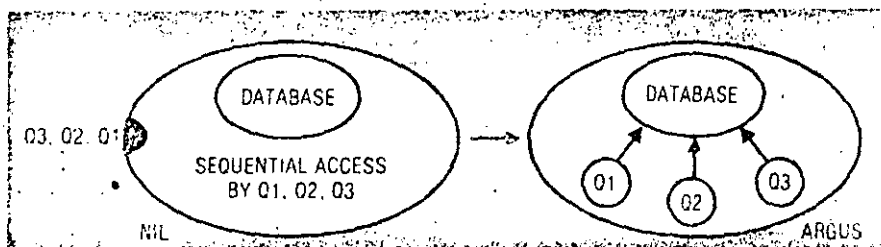


Figure 6. Optimization of distributed sequential processes.



is destroyed by a crash, and permanent storage, which may be recovered with high probability. Argus allows user to specify the granularity of recovery by atomic actions, which may abort or commit. The mechanism of atomic actions is used to recover from software failures and to define the granularity of transactions that need to be atomic in maintaining consistent states.

Recovery in NIL is transparent to the user. The system automatically takes periodic checkpoints and uses an optimistic recovery technique to recover from failures. Atomic actions are therefore not needed to recover from hardware failures. Software failures are handled by the NIL exception mechanism. All statements (including remote procedure calls) are atomic actions. A separate mechanism for defining atomic actions is therefore not needed.

Delegating responsibility for concurrency and recovery to the system makes the resulting language higher level. But it requires a different model of the problem being solved and different language mechanisms to achieve the desired computational effect. For example, Argus handles hardware failures, software failures, and atomic transactions by the same language mechanism (atomic actions). NIL handles hardware failures transparently, software failures by the exception mechanism, and atomic transactions by serial processes, demonstrating a very different software design approach.

Recovery and internal concurrency are only two of many design issues for distributed processes. For example, distributed systems should have mechanisms for the dynamic creation and dynamic linking of processes so that they may evolve during execution. Mechanisms are needed for communicating with external devices, including external distributed processes with different hardware and software characteristics. A robust technology for distributed software components must transcend traditional programming language concepts and incorporate

operating system and communication technology.

### Objects, classes, and hierarchies

- (1) A thing that can be seen or touched, material that occupies space;
- (2) a person or thing to which action, thought, or feeling is directed;
- (3) what is aimed at: purpose, goal, end.

—Definition of "object,"  
*Webster's Dictionary, 1980*

The term "object" has become a ubiquitous buzzword that was independently adopted by the operating system, programming language, and database communities to denote software components having a hidden state and a set of operations or capabilities for transforming the state. Both data and process abstraction are "object-oriented," but the term has been most closely associated with Smalltalk,<sup>7</sup> a programming language developed at Xerox PARC that superimposes hierarchical inheritance on data abstraction.

The set of classes of Smalltalk is organized as a tree structure with a root class called "object" containing properties possessed by all objects, such as the method "copy" for creating instances of any object. Subclasses are a specialization of the parent class that possesses all of its properties as well as properties special to the subclass. Thus the class "vehicle," with methods "weight" and "owner" applicable to all vehicles, could have a subclass "car" with method "passengers" and a subclass "truck" with method "capacity." The subclass "car" could in turn have subclasses "Buick" and "Toyota."

By associating the superclass "car" with the subclass "Toyota," we permit callers of objects of the class "Toyota" to use methods and variables of the superclasses "car," "vehicle," and "object." A new class can simply specify incremental attributes and reuse attributes that the new class shares with already defined classes. Class hierarchies determine a capital-intensive paradigm for the flexible reuse of already defined data and

#### Smalltalk

Objects in Smalltalk represent their internal state by "instance variables," and have operations called "methods" which are invoked by "messages" from other objects. Messages specify the name of the object being called, the name of the method to be invoked, and actual parameters of the invoked method. They are like procedure calls of conventional languages, but binding of the method name in the message to the method actually invoked occurs at execution time rather than at load time.

Objects are created from "class definitions" which specify methods and instance variables common to objects of a class. Classes correspond to types in traditional languages. Classes in Smalltalk may inherit instance variables and methods of a superclass. The complete set of variables and methods available in a class includes not only those directly defined in the class but also those defined in superclasses of the class.

program behavior in defining new system components.

There is a difference between inheriting attributes from a superclass and importing attributes from a global environment. Imported attributes may be invoked from within an object but cannot generally be called by users of the object. Inherited attributes can be directly invoked by callers. They can be exported to and inherited by callers. They become "owned" by the objects that inherit them, and may be disposed of by their owners in any way that is deemed desirable. Inheritance is transitive in the sense that if *P* inherits *Q* and *Q* inherits *R* then *P* inherits *R*, while importing is not transitive.

In the terminology of ports, imported attributes determine additional output ports, while inherited attributes determine additional input ports callable by anyone that knows the name of the object. Introducing inheritance for distributed processes requires augmenting the set of input ports to include ports for inherited attributes.

### Inheritance

The importance of multiple inheritance as a mechanism for system evolution is underlined by the fact that natural inheritance is based on the genes of two parents rather than one.

The Smalltalk paradigm of object-oriented programming is spreading to other language cultures. Class hierarchies of the Smalltalk variety have been used as a basis for extending a number of existing programming languages, and have resulted in Clascal (Pascal with classes) and C++ (C with classes). Lisp, which has a simple

core that lends itself to extensions, has been extended to support classes, called flavors in the Lisp community.<sup>8</sup> Lisp flavors may "mix in" methods and variables from more than one superclass, resulting in multiple inheritance of the attributes of several superclasses by a newly defined class. (The term "flavors" derives from Steve's ice cream parlor where nuts and chocolate pieces can be "mixed in" to flavor a dish of vanilla.)

Inheritance may both enrich a class with extra features and specialize a class to perform a particular function. Consider for example a class "window," whose instances are windows on the screen of a personal computer. Windows have basic attributes such as a location on the screen. They may be enriched by inheriting attributes such as "border," "label," and "scroll." They may also inherit attributes such as "Lisp" or "Pascal" that specialize the window to a particular language. As shown in Figure 7, enriching features may be "mixed in" to the class window without restriction, while features that specialize a window to a particular language are incompatible with each other. Thus inherited attributes may be subject to integrity constraints similar to those for databases. In its most general form an inheritance structure is a relation among templates that is automatically acquired by all instances as they are created.

An inheritance structure on software components mirrors the growth of knowledge in a database by building on what already exists rather than by starting from the beginning for every software component. Inheritance of attributes by software components is analogous to inheritance of acquired capital resources

and inherited genes from one generation to the next in human societies. But inheritance of software components is more flexible because a component can be bound to different ancestors on different instances of execution. In some object-oriented systems, the binding may be changed even during execution. This corresponds to genetic engineers changing the ancestors of a person after he has been born.

Object-oriented languages of the Smalltalk variety have the following characteristics:

(1) Objects with operations on an internal state are the primary software component. The internal state persists between successive operations and may, for long-lived objects, change its set of applicable operations while maintaining the identity of the object.

(2) Operations and states may be inherited from previously defined classes by single or multiple inheritance so that new functionality may be incrementally defined in terms of previous functionality.

The first of these characteristics defines the essence of being object-oriented. The second defines an attractive paradigm for organizing a library of classes so that new classes of objects can build on the properties of previously defined classes.

Class inheritance can be viewed as a structural relation of a library of components. Classes in Smalltalk can be thought of as library components, since they can be repeatedly reused either to create objects or as superclasses of lower-level classes. The class hierarchy of Smalltalk determines a tree-structured rather than a flat library. Class hierarchies provide a structured way of organizing components in a library so they can be systematically reused.

### Library structure and design

Libraries promote the reuse of existing knowledge in the creation of new knowledge.

Program libraries are databases of reusable software components.

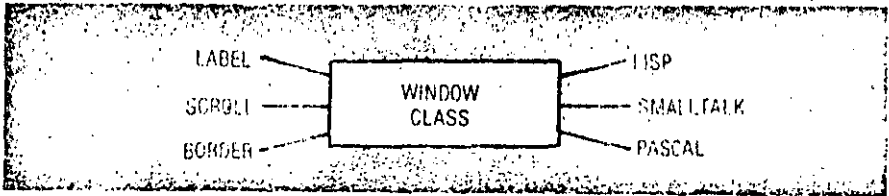


Figure 7. Inherited enrichment and inherited specialization.

libra  
near  
mp  
create  
shoul  
poner  
know  
respe  
or fil  
conve  
custe  
peop  
com  
must  
1990  
gent  
about  
struc  
libra  
In  
folle  
(1  
the  
(2  
mai  
(3  
(4  
link  
(5  
inse  
(6  
por  
(7  
ne  
bui  
lib  
(8  
lik  
on  
na  
fer  
Fe  
rel  
pli  
pr  
(9  
su  
ta  
de  
m  
cl  
P  
th  
cu  
d

Library design is concerned with organizing collections of software components so they can be easily created and used. Program libraries should contain not only software components but also tools for organizing knowledge such as catalogs. In this respect they are like libraries of books or films. However, they differ from conventional libraries in that their customers are computers as well as people. Library tools for welding components together into programs must, therefore, be automatic. By the 1990's we are likely to have "intelligent libraries" that use knowledge about application domains in constructing composite programs from a library of software components.

In designing a program library the following issues must be considered:

- (1) What kinds of components can the library contain?
- (2) What is the granularity and domain of application of the library?
- (3) What kinds of clients (programs, people) will use the library?
- (4) How are components loaded, linked, and invoked?
- (5) How are components created, inserted, inspected, retrieved?
- (6) What relations among components may be expressed?
- (7) What kind of knowledge is needed to aid programmers in building composite programs from libraries of software components?

In early programming languages like Fortran, subprograms are the only library components, although named common data blocks are effectively a second "data library." Fortran libraries are flat and require relations among programs to be implicit in the calling structure of subprograms.

In Ada, library components may be subprograms and packages but not tasks. Library structure is flat; but dependencies among components must be explicitly specified by "with" clauses. In Smalltalk, library components are organized into a hierarchy that allows new knowledge to be incrementally added to the library database. Programming environ-

ments such as Unix may be viewed as libraries that support exceptionally diverse sets of clients and granularity within a single system. More generally, any database may be viewed as a library of persistent components whose lifetime is longer than the operation which accesses them.

Libraries may be characterized both by the kinds of components they contain and by the way that human and computer clients of the library use these components. Human clients include managers, analysts, and programmers with different needs and expectations. Computers also have multiple interfaces with software components, including compiler interfaces for program creation, tool interfaces for debugging, version control, and execution-time interfaces for module execution.

Libraries define a collection of resources external to a given component that complements the resources built up within a component while it

computes. In block-structured languages the distinction between internal and external resources is coupled with the distinction between dynamic and static creation. Internal resources represented by local variables are created dynamically on entry to the block in which they are declared, while external resources represented by components are fixed prior to execution. Distributed-processing languages and operating systems generally allow execution-time creation and linking of components and thereby encourage the programmer to think of components as first-class objects whose properties are not dissimilar from properties of variables.

The extension of inheritance to concurrent and distributed processes is an interesting library design issue. Inheritance is essentially a block-structured compile-time mechanism for constructing types out of previously defined types. In the world of Lisp and of dynamically linked

### Dynamic linking

One aspect of the execution-time environment is the loading and linking of library components. Languages like Fortran and Ada have a loading and linking phase prior to execution and require the set of components and their bindings to each other to be invariant during execution. Loading and linking are relegated to an operating system whose operation is not under the control of the programmer.

This limitation is unduly static and makes an artificial distinction between program evolution during program development and during program execution. An alternative is to allow loading and linking to be performed by programming language commands during program execution. This approach is taken in NIL, which allows processes to be loaded dynamically by executing a create instruction, and which also allows ports of a created process to be dynamically connected to ports in other processes both at process creation time and during subsequent execution of the process. The processes of a NIL computation may include compilers which compile new processes and add them to the library concurrently with the execution of other processes, so that they become available dynamically to processes already executing. Moreover, any given process of the library can have multiple instantiations, each with a different set of port connections to other processes.

This dynamic flexibility in creating and linking processes contrasts sharply with the static relations among components in Pascal and Ada. Ada was carefully designed to support evolution of programs during program development, but contains no provision for evolution of programs during execution. Languages like NIL, on the other hand, are designed to support program evolution during both program development and execution. They attempt to combine the dynamic advantages of languages like Lisp with the static advantages of strong typing. They provide an extra dimension of reusability which may well be an important factor in creating an effective software technology for the development of long-lived multi-module programs.

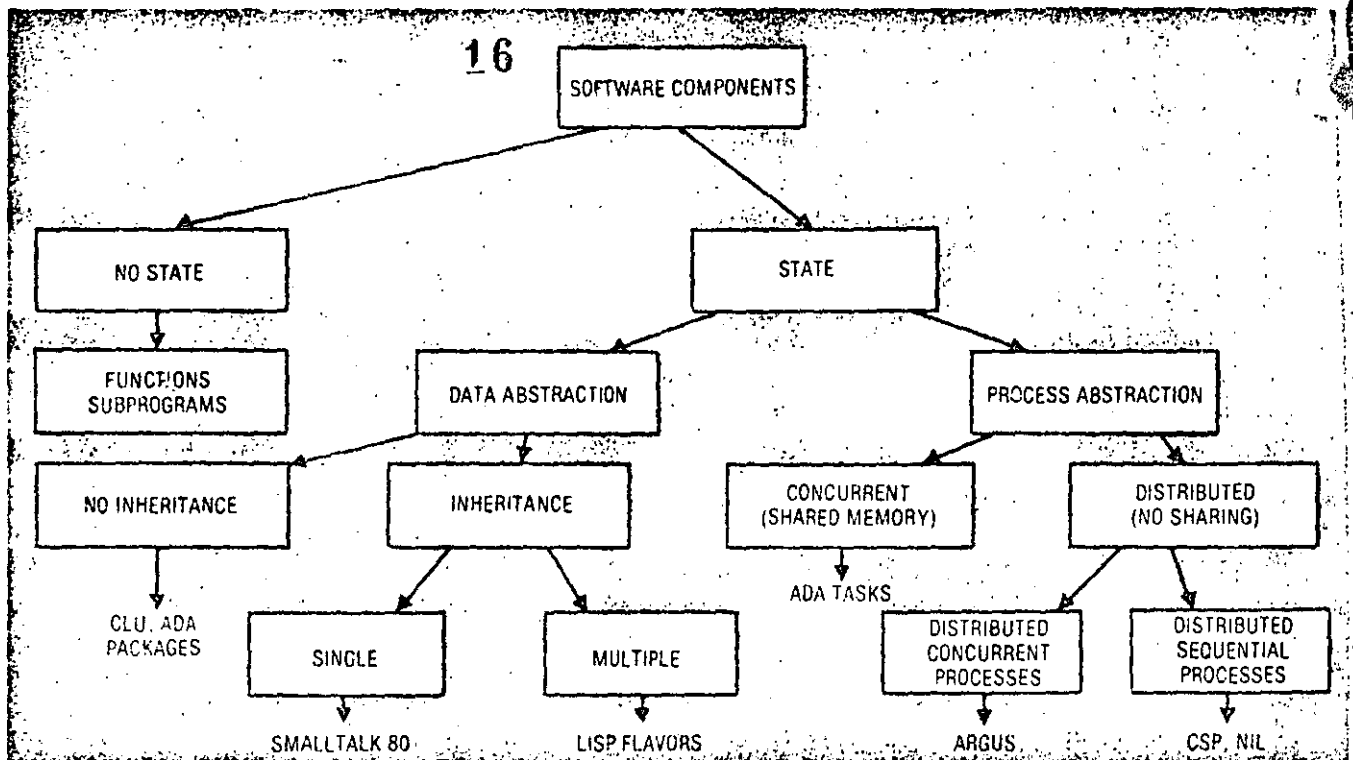


Figure 8. Taxonomy of software components (with examples).

distributed processes, access rights to components are first-class objects and can be passed between processes dynamically. Dynamic passing of access rights is more flexible than the inheritance in Smalltalk but also requires more work on the part of the programmer. The flexibility of being able to inherit attributes dynamically can be important in long-lived systems that evolve during execution.

### Taxonomy of software components

#### Classified information?

Software components may be classified as in Figure 8. This taxonomy identifies state, inheritance, concurrency, and sharing as discriminating characteristics for distinguishing among the software components of languages. It provides little guidance in classifying the large number of languages whose primary components are functions and subprograms (Fortran, Lisp, APL, Prolog) because they differ in their mechanisms for compu-

tation rather than inter-module communication. But it provides a rather satisfying, simply structured classification for languages with data and process abstraction and highlights key differences in the properties of their components.

The separation between data and process abstraction is based on the intuitive notion that data abstractions are passive while process abstractions are active. All operations in a data abstraction may be passively accessed at any time. Process abstractions control both when they accept an operation (by an input queue) and where they accept an operation (by an accept statement). Protected data abstractions, such as monitors, fall between data and process abstraction because they control when but not where they can be accessed. Monitors are generally implemented as specialized processes since the monitor queue is subject to precisely the same scheduling rules as queues associated with input ports of processes.

Our taxonomy of software components is tentative and shows that

software components technology is in a state of transition. Programming languages like Ada were designed as sequential languages and have concurrency as a special feature, not properly integrated into the language. As concurrency becomes the norm, process abstraction will become the central abstraction mechanism, and both data and function-abstraction will be defined as specialized forms of process abstraction. When this happens, data abstraction may well disappear as a concept distinct from process abstraction.

Our analysis suggests that the user model of computation for process abstraction will be that of distributed sequential processes. This requires the user to be aware of concurrency between distributed processes, while concurrency within distributed processes (if any) is hidden from the user and introduced for purposes of optimization by the system. Sharing of data by concurrent processes is likewise hidden from the user and managed by the system.

# SOFTWARE MARKETING EXECUTIVES

## TODAY,

There are thousands of software products for  
buyers to choose from . . .  
no effective way to compare and evaluate . . .

A new software marketing concept is needed . . .  
. . . To place your new software products in front of  
lots of qualified buyers.

. . . To lower your marketing costs by 50% or more.

. . . To provide you with the volume buyers of a  
trade show, in a modern personal-sales oriented  
environment.

## TODAY,

We can help you solve all of your software  
marketing problems . . .

and move your products quickly at low cost.

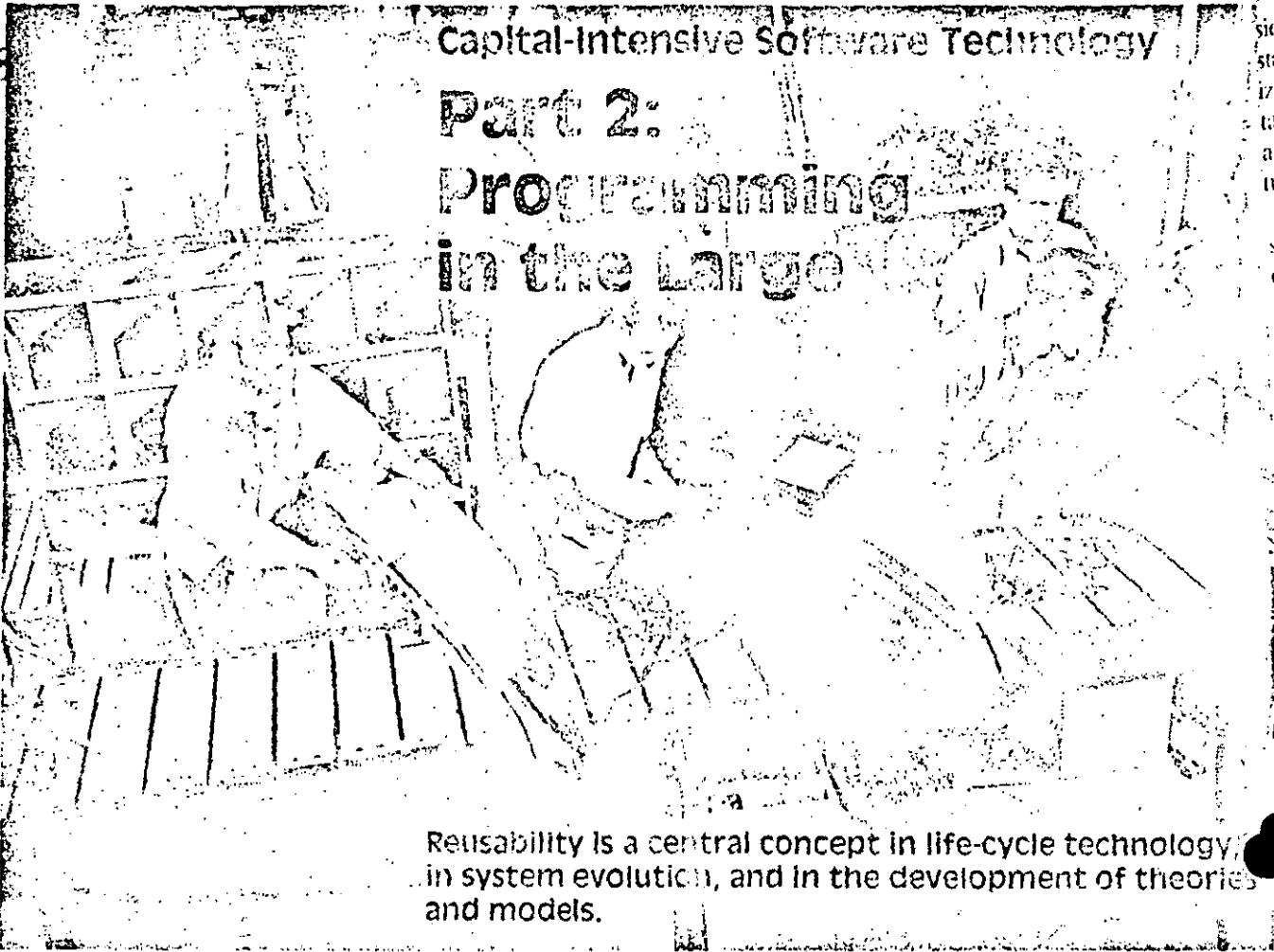
A new computer software marketing channel  
will be ready to start working for you early  
in 1984.

For reservations and information,  
call (213) 385-5118.

**DO IT NOW!**

**COMPUTER  
TECHNOLOGY  
CENTER**

... 1625 W. Olympic Blvd., Los Angeles, CA 90015



Capital-Intensive Software Technology

Part 2:  
Programming  
in the Large

Reusability is a central concept in life-cycle technology in system evolution, and in the development of theories and models.

Changing paradigms of software technology

The change from a program-centered to a data-centered view of programming is comparable to the shift from the earth-centered to the sun-centered view of the solar system brought about by the Copernican revolution.

Charles Bachman, Turing Lecture, 1973<sup>9</sup>

As a first approximation, the evolution of software technology can be characterized by the following phases:

- 1950's—stand-alone programs (transient data, subprograms),
- 1960's—operating systems, databases (managing banks, airplanes),
- 1970's—software engineering (life cycle, abstraction, methodology),
- 1980's—interface technology (personal computers, modular languages), and
- 1990's—knowledge engineering (intelligent components, for computers, for people).

In the early days of computing the paradigm for programming was

writing subprograms that realized algorithms rather than modeling complex evolving real-world systems. Emphasis was on computations with transient data structures that could be discarded once the computation was completed. Programs were the primary capital goods and data was a "consumer good" in the sense that it was supplied as input and returned as output to the consumer. Subprogram libraries for common algorithms were regarded as the principal capital-intensive mechanism for reducing the programming effort.

As applications became larger and more ambitious, their nature changed to data management and embedded computing applications with nontran-

sient data structures representing the state of an evolving system or organization. Data became a primary capital resource since programs often had shorter lifetime than the data structures on which they operated.

Embedded computer applications, such as those that control banking operations, airline reservations, or aircraft flight, require two databases—one for the application and the other for program development. The application database contains both permanent and transient facts about the domain of discourse, while the program development database includes current and old versions of software components as well as specialized tools for testing or otherwise manipulating the components.

By 1970, perceived similarities between constructing large software systems and large physical structures, such as bridges and buildings, led to the birth of the discipline of software engineering. The life-cycle model caused attention to shift from software products, such as individual subprograms, to the process of software development.

The personal computer revolution of the 1980's has ushered in a new technology of man-computer interfaces. It involves the use of high-resolution multiple-window screens that simulate multiple piles of papers on a desktop. Multiple views of programs and data can be handled well by the emerging interface technology. The improved man-computer interface technology is being supplemented by an improved intermodule interface technology based on a distributed rather than a block-structured model of computation. Its aim is to provide a sounder framework for software components technology than that provided by current block-structured languages.

By the 1990's, interface technology will have become sufficiently capital-intensive to increase our system-building capability by several orders of magnitude. Ambitious software systems that failed in the past will become technologically feasible. For example, computer-aided instruction,

which received a bad name in the 1960's and the 1970's, is likely to become cost-effective in the 1990's; it could materially change the style and pace of learning in schools and universities. Flat-screen technology could cause paper books to be replaced by much more versatile computer books. Knowledge databases could play an active role in amplifying our mental abilities, both in everyday activities and in research that extends the frontiers of knowledge.

### Life-cycle paradigms

**Life-cycle models provide a uniform framework for problem solving within which reusable methodologies and tools can be developed.**

Attempts to understand the process of program development have led to a progression of life-cycle models including the static waterfall model, the more dynamic operational model, and the futuristic knowledge-based model.

### Fifth-generation computers

The Japanese Fifth-Generation Computer Project is an attempt to create a capital-intensive technology for knowledge engineering.<sup>10</sup> Its central theme is to add intelligence to the high-bandwidth interface technology of fourth-generation personal computing systems. Its proposed architecture includes a database machine and a problem-solving and inference machine. Its proposed system programming language is the logic programming language Prolog. Its software includes support for natural language and speech understanding and problem solving over a wide set of problem domains. The project includes not only technical goals such as increasing productivity and saving energy, but also social goals such as coping with an aging society. Some researchers regard the project as overambitious. But it has a worthwhile set of goals, which, even if not achieved in their entirety, can catalyze an integrated research effort that could give Japan a technological lead in developing computing systems for the 1990's.

The basic premise of the Japanese Fifth-Generation Computer Project is that our prime concern in the 1990's will be the processing and management of knowledge. A logic-based language was chosen as the system programming language because logic was perceived to be the basic tool for managing and manipulating knowledge. The hardware emphasizes the management of knowledge databases, the software emphasizes problem solving and inference, and the user interface emphasizes knowledge acquisition by understanding natural language inputs.

The Japanese have done the computing profession a service by presenting their vision of an integrated knowledge-engineering environment in such a clear and public manner. This has placed the goal of achieving such an environment squarely in the public domain. But the Japanese have a clearer vision and are pursuing the goal with greater single-mindedness than other nations and may therefore be the first to achieve the goal, with all the commercial and other advantages that this entails.

(1) The waterfall model: requirements — design — implementation — maintenance. In the waterfall model the development of software proceeds through a number of stages. Each stage has documentary output that serves as the input to the next stage. Early stages specify an informal behavioral abstraction of *what* is computed. This abstraction is progressively refined into a formal implementation of *how* the behavior can be realized. Maintenance and enhancement is performed on the implemented program.

(2) The operational model: executable specification — transformations — efficient implementation. In the operational model software development proceeds from an executable problem-oriented specification (rapid prototype) through a sequence

of transformations to a more efficient implementation-oriented realization.<sup>11</sup> Early stages are independent of computational resources. Transformations from the problem-oriented specification to an efficient implementation are automatic wherever possible. Maintenance and enhancement changes are performed on the problem-oriented specification, which is then optimized.

(3) The knowledge-based model: project database, knowledge-based assistant. In the knowledge-based model software development is under the control of a knowledge-based activity coordinator, which coordinates access by multiple developers and users and logs the states and history of all information in the project database. The computer is an active partner in the process or program

development. The model provides a framework for automating a variety of life-cycle models, including the waterfall and operational models. However, it favors life-cycle technologies that can be automated.

**Waterfall model.** The waterfall model was developed in the late 1960's—before knowledge-based automation—and is viable with a non-computerized project database. We now have considerable experience and data for projects using this model. For example, Boehm<sup>12</sup> draws on a database of 63 projects in developing a constructive cost model—Cocomo—for estimating levels of effort and time schedules for software projects. His results show that the uniformities hypothesized by the model do in fact exist. In spite of such successes, however, the waterfall model has the following drawbacks.

(1) It is geared to program development by humans rather than computers.

(2) It does not provide feedback concerning requirements and design behavior till late in the implementation phase.

(3) Documentation is manual, voluminous, static, and incomplete.

(4) Maintenance and enhancement is performed on low-level, already-optimized implementations rather than on problem-oriented specifications.

(5) The behavioral abstraction paradigm with its inflexible separation between behavior and implementation may be inappropriate for automatic life-cycle management.

The waterfall model embodies development procedures prevalent in the 1960's, before the potential of the computer as an active partner in program development was fully understood. The manual nature of requirements, design, and implementation activities requires the system designer to make the major system-development decisions and cast them in concrete without any feedback from the computer. Voluminous documentation of static system structure at successive points in system evolu-

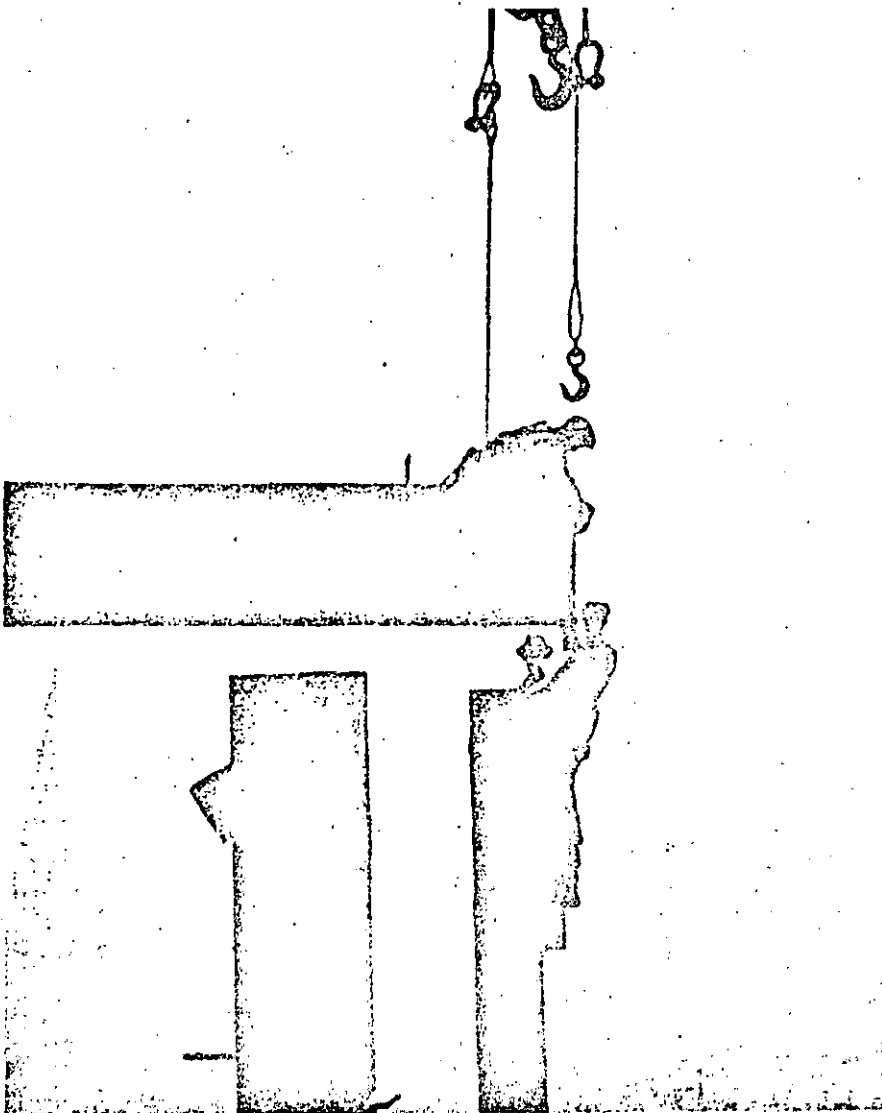


Photo by Lewis W. Hine courtesy of the film *America and Lewis Hine* (See p. 42.)



tion is required, leaving little energy for documenting the process of program development. Moreover, program development is regarded as complete once an implementation is delivered. Maintenance is performed by patching the implementation without any redesign or redevelopment.

Life-cycle automation requires an underlying abstraction paradigm that is susceptible to automation. The traditional behavioral abstraction paradigm of the waterfall model identifies abstraction with external behavior and implementation with internal code structure. Program development by refinement of behavioral abstraction is not easily automated. Formal specifications and verification that specifications realize intended behavior are motivated by analogies with mathematics and reflect a manual rather than an automated view of the "meaning" of program modules. The complexity of specification and verification systems provides a warning that this form of abstraction may be unnecessarily complex, and suggests that alternative abstraction paradigms should be sought.

**Operational model.** One alternative abstraction paradigm is the operational life-cycle model. This model is based on operational rather than behavioral abstraction. An operational abstraction of a system or module captures its behavior by an executable specification in terms of a problem-oriented model of computation. Instead of hiding the implementation mechanism, an operational abstraction explicitly uses the implementation as a basis for specifying behavior. An executable specification is as formal as a behavioral specification, but its behavior is specified implicitly (and automatically) by its set of executable computations rather than by an implementation-independent formal specification.

Operational abstraction determines a process-oriented paradigm for problem solving. Instead of verbal requirements and a design that determines the modular structure of the implementation, an executable

model for an idealized computing environment is developed directly and tested by a high-level interpreter. This provides early feedback to both the end-user and system designer on the functionality of the intended system, and it serves as a basis for developing an optimized, acceptably efficient realization of the prototype.

Transformation of an executable specification into an acceptably efficient realization is a complex task that requires human decisions as well as automated implementation of the consequences of human decisions. But this task is more amenable to automation than the classical program development paradigm because we start from a formal executable system that captures program behavior by a process-oriented abstraction.

The operational life-cycle model is competitive with the waterfall model even in a manual environment because of the feedback provided by executable specifications. The payoff of the operational model lies in its greater compatibility with automated program development.

**Knowledge-based model.** But there is still another abstraction paradigm: the knowledge-based model. Knowledge-based program development systems based on the operational life-cycle paradigm have the potential of increasing software productivity by several orders of magnitude.

The knowledge-based model is effectively a specialization of the artificial intelligence paradigm of expert systems applied to the domain of program development. It needs three kinds of databases.

- A general-purpose, domain-independent environment.
- A special-purpose, project-oriented set of tools.
- A project database with multiple versions of application programs.

Knowledge of the programming process and of the program development process is encoded into a collection of expert systems that make use of knowledge in each of the three databases. The expert systems are called upon either explicitly by the

programmer or implicitly during the performance of a program development task.

The development of a knowledge-based software assistant is a very ambitious task whose realization has been estimated to require 15 years of cooperative effort by several institutions.<sup>13</sup> However, it provides a framework for a quantum leap forward in capital-intensive software technology. Moreover, expert-system technology has reached a level of maturity that should be capable of sustaining systems of this magnitude.

### Maintenance, enhancement, and evolution

Plus ça change, plus c'est la même chose. (The more it changes, the more it stays the same.)

Complex structures, both natural and social, are generally the result of

### Operational and behavioral thinking

The distinction between operational and behavioral thinking is not restricted to life-cycle models. It arises in many areas of computer science and also in mathematics and physics. Operational semantics differs from denotational and axiomatic semantics. Mathematics has a behavioral view of model building and abstraction. The difference between mathematical and computational views of the world is essentially a difference between behavioral and operational approaches to model building. In physics, relativity theory and quantum theory are operational models because they take operations performed by an observer into account in formulating the model. Operational models of physics may be contrasted with the Platonic notion of absolute space and time that underlies Newtonian mechanics. Note, however, that operational semantics in computer science requires semantics to be defined in terms of the internal structure of states while operationalism in physics explains natural phenomena in terms of the way they are observed and has a distinctly operational flavor.

### Evolution of organizations

The evolution of software systems may be viewed as a special case of the evolution of organizations with both human and computer components. There is a considerable literature on the structure, social dynamics, and adaptability of organizations. At a very general level, Toynbee's pessimistic study of the genesis, growth, breakdown, and disintegration of civilizations is about the failure of organizations to adapt to changing environments, and suggests that radical renewal must supplement gradual change as a mechanism for system evolution.<sup>15</sup> The text *Organizations*, which is a source book for Simon's Nobel-prize winning work on formal models of organizational behavior, is a good starting point for readers interested in this area.<sup>16</sup> *Organization Development* presents an analytical framework for the development of organizations in terms of flows of information among their components.<sup>17</sup> Holland explores the problem of adaptation for both natural and artificial systems, emphasizing the response of such systems to changing environment.<sup>18</sup> The similarity of models of large industrial organizations and large computer systems is reflected in the computer literature in anthropomorphic terminology such as "actors" and "messages" in the modeling of "societies" of interacting computer programs.<sup>19</sup> Milner's *A Calculus of Communicating Systems* is an example of a formal (algebraic) model of communicating systems whose components may be people or computers.<sup>20</sup> The study of evolutionary behavior of mixed man-computer systems, and of interfaces that allow creativity and growth of people in a computer environment, is central to the development of a reusable software technology that combines current efficiency with adaptability to change.

evolutionary development rather than a single creative act. This is also true of large software systems. A new programming language such as Ada evolves from experience in the design and implementation of previous languages such as Pascal, and in turn forms a basis for the design of future languages. Successful programming systems such as Unix evolved from small beginnings. They achieved their success by having a simple core to which facilities could easily be added.

#### Maintenance and enhancement.

Large software systems must be constructed so that modification and evolution can be accomplished in a time proportional to the magnitude of the changes rather than to the size of the system.

Large systems that are easily maintained and enhanced have the property of local modularity and are constructible in an evolutionary manner from primitive components. They can be constructed by incremental "builds" of subsystems, which can be independently tested and verified. Therefore, an adequate solution of the maintenance and enhancement problem depends on a evolutionary system structure not only for a large system as a whole but also for its component subsystems. Maintainable systems require not only static modularity but also "dynamic" modularity that facilitates incremental development, testing, and rapid prototyping. Solving the maintenance problem requires an evolutionary life-cycle methodology that allows complex systems to evolve from a simple core by multiple independent extensions. Failing to find a simple expandable core may result in system rigidity even if the system has a high degree of modularity. So the additional requirements on modularity needed to support evolutionary development are an important area of study.

Iterative enhancement<sup>14</sup> is an example of an evolutionary life-cycle approach. It advocates using a skeletal implementation (rapid prototype) as a starting point for iterative redesign of what has already been produced and

evolutionary addition of new features until the system is completed. When the set of tasks needed to complete project can be predicted, they can be listed in a project control list and systematically scheduled. The approach is useful even when the set of subtasks and the end result are incompletely defined. For example, the present article was developed by iterative enhancement of an incomplete specification, starting from a brief discussion of the capital-intensive nature of Ada and growing by iterative revision and expansion to its present scope and size. The technique of iterative enhancement was first developed in the context of software engineering but is just as pertinent to the writing of articles and books, where evolution is an inherent part of the process of creation. Text-editing systems and other computerized knowledge-engineering aids greatly facilitate evolutionary development of manuscripts and are having a profound impact on the writing habits of both technical and nontechnical authors.

**Evolutionary flexibility.** Lack of evolutionary flexibility contributed to the failure of technologically advanced countries like Great Britain in coping with competition from countries whose industrial development occurred later in time. It could similarly lead to dissipation of the current US lead in the software field to countries like Japan whose software technology is less dependent on old software systems and management structures. Inability to adjust to changing technology was a cause of great pain and social dislocation in the industrial revolution. Evolutionary flexibility, both for individuals and for the technology as a whole, should be a primary goal of information technology.

There may be a distinction between "evolution in the large" for very large organizations with long time horizons measured in decades or centuries and "evolution in the small" for smaller (but still large) organizations with time horizons measured in months or years. Adaptation to changing technology is concerned

tures  
When  
ete a  
in be  
l sys-  
oach  
asks  
etely  
arti-  
en-  
cifi-  
sion  
Ada  
and  
size.  
re-  
on-  
just  
cles  
in-  
on.  
om-  
ids  
op-  
g a  
bits  
cal

of  
to d-  
in  
n-  
c-  
y S  
y  
-  
1

Photo by W. Hine courtesy of the Ilim America and Lewis Hine (See p. 42.)



23

with evolution in the large, while development, maintenance, and enhancement of a particular system is concerned with evolution in the small. Tuning of a system for a particular set of tasks and time horizons may increase its cost and reduce its efficiency for narrower classes of applications and constrain its adaptability for broader classes of applications.

Maintainable systems should be modular both "in the small" to allow modification of functional components in specific applications and "in the large" to allow changes in major components of the technology, such as the change from sequential to concurrent programming or from central processing units to distributed processing hardware. Systems should be designed not only for evolution in the small for short time horizons, but also for evolution in the large for long time horizons, so that they are robust with regard to extension in both space and time.

The problem of evolutionary flexibility arises in its most acute form in the context of adaptation to a changing technology. It is a bottleneck in the adaptation of embedded systems to changing environments, since maintenance and enhancement account for 80 percent of total life-cycle costs. Adaptive systems that can acquire and subsequently use knowledge, such as expert systems or theorem provers, must have databases designed for evolution.

Evolution involves reusability in response to change. Thus, evolutionary systems are capital-intensive according to our definition. Evolution, adaptation, and maintainability are additional synonyms for reusability.

### Reusable concepts and models

Research in any discipline is judged largely by the degree to which its products are reusable.

**Reducibility and equivalence.** Reducibility of a problem *B* to a problem *A* means that the techniques used in solving problem *A* may be reused in solving problem *B*. Reducibility is thus a synonym for reusability. A compiler reduces programs in a problem-oriented language to equivalent programs in an executable machine language (a language for which the program execution problem is already solved). The reducibility of the context-free grammar parsing problem to the matrix

multiplication problem permits the algorithm for matrix multiplication to be reused in parsing a context-free grammar. All reducibility results are effectively equivalence-preserving mappings of a problem from a less familiar form to a previously encountered form.

Equivalence is effectively two-way reducibility. Having proved  $B$  equivalent to  $A$ , we can reuse all of  $A$ 's known properties in talking about (or executing)  $B$  and vice versa. In defining any equivalence class, we are effectively factoring attributes of the domain of discourse into primary attributes that are reusable for all elements of the class and into secondary attributes that may, for the time being, be ignored. Different choices of equivalence class determine different abstractions with different choices of what is regarded as fixed or variable. For example, program-centered and data-centered programming can be distinguished by differences in the choice of equivalence relations that determine primary and secondary attributes.

By defining equivalence-preserving transformations we can navigate within an equivalence class and examine secondary attributes for invariant primary attributes. For example, transformations on equivalence classes of functionally equivalent program specifications allow us to navigate from a program representation suited to human understanding to a representation suited to computer efficiency. Life-cycle models of program development provide a framework for navigation within an equivalence class from a requirements specification through a design specification to an implementation that permits flexible enhancement. One of the objectives of expert systems is to codify knowledge about such equivalence classes to facilitate automatic goal-directed equivalence-preserving navigation.

**Models.** Model building is closely connected with reusability. Mathematical logic is concerned with the development of models of valid reasoning that may be reused for all

24 Interpretations of nonlogical symbols in specific domains about which we wish to reason. The first-order predicate calculus is a framework for reasoning about mathematical objects such as sets, functions, and predicates. Systems of computational logic such as dynamic and temporal logic specialize the predicate calculus to permit reasoning about domains of computational objects such as programs. Systems of modal logic allow reasoning in situations where the "modality" of an assertion may be other than conventionally true or false (for example, there may be quantitative or qualitative probabilities of being true).

Semantic models provide a framework for expressing the meaning of programs and programming languages in terms of operational semantics, which models the process by which results are computed, or denotational semantics, which attempts to model meaning by abstract mathematical denotations. Denotational models are less dependent on a particular model of computation than operational models. But operational models provide insight into the model of computation and may be useful to designers and implementors in understanding what actually happens during execution. The greater abstraction of denotational models provides greater potential reusability, but they may require greater overhead of interpretation and provide less intuitive understanding than operational models. If reusability is to be enhanced, the choice of an abstraction to model the meaning of computational constructs should be governed by the purposes for which the abstraction is to be used.

**Invariance and regularity.** Discovery of invariants is a basic paradigm for both mathematical and computational abstraction. The input-output relation realized by a function is an invariant for all programs that realize the function. Invariants of the program state during execution are used as a basis for program verification. A program  $P(x)$  is

### Specialization—the converse of abstraction

An abstraction characterizes a class of phenomena by their common (invariant) attributes and hides (ignores) distinguishing attributes of instances that are not common to the complete class. Greater abstraction results in greater reusability and reduces the costs of problem solving by distributing the cost of developing the abstraction over its uses, but the converse process of making distinctions between instances of an abstract class is also an essential ingredient in problem solving. The lawyer thrives on distinguishing new cases from precedent-setting abstractions established by previous cases. Good information engineering requires shedding excessive generality by making distinctions that provide new insights or allow efficient solution of the problem at hand.

The converse of the process of abstraction may be referred to as specialization. Specialization may be technically defined in terms of contexts which constrain the generality of the abstraction and may be used as a basis for optimization.<sup>21</sup> The formal notion of specialization has been applied to toy problems, such as specializing the "reverse list" function to finding the last element of the reversed list, and to more substantive problems such as the derivation of algorithms for context-free grammars and graph problems. Examples of the practical use of specialization include supplying parameters to procedures, constraining a type to a subtype in order to realize greater space or time efficiency, selecting an alternative in a menu-driven graphics application, zooming in to obtain more detail in a graphics display, and specializing a requirements specification to a particular program design.

Since specialization is as ubiquitous and important a concept as abstraction, mechanisms for specialization deserve to be classified and formalized as rigorously as those for abstraction. Schemes for navigation within an equivalence class should make equal use of controlled specialization and abstraction as central tools in problem solving.

sym-  
which  
order  
rk for  
I ob-  
and  
tional  
poral  
culus  
ns of  
pro-  
llow  
the  
y be  
e or  
be  
ob-

me-  
g of  
an-  
nal  
ess  
or  
at-  
act  
ta-  
n a  
on  
ra-  
he  
be  
rs  
P-  
er  
ls  
/  
e  
n  
s  
1

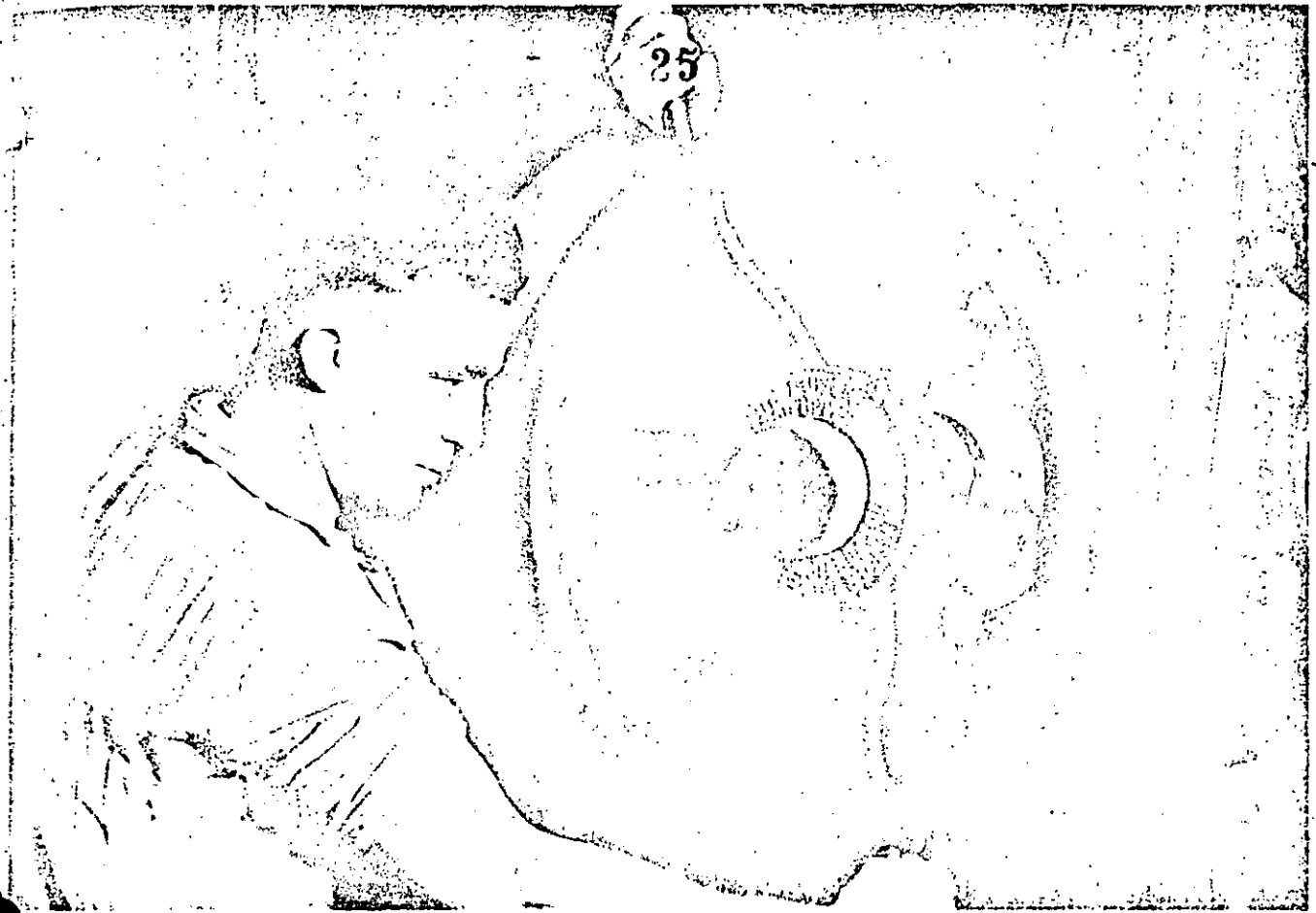


Photo by Lewis W. Hine courtesy of the film America and Lewis Hine (See p. 42.)

an invariant that determines a uniform rule of computation for all arguments  $x$  in the domain of  $P$ . The programming process involves the creative discovery of invariants during the specification, verification, and realization of programs so that demonstrably correct programs that realize their specification can be developed. Many of the practical tasks in software engineering can be formulated in terms of the discovery of reusable invariants in a class of computations or class of concepts. The notion of invariance is, therefore, yet another synonym for reusability.

Pattern recognition and regularity are two further synonyms for reusability that are closely related to invariance. Pattern recognition is concerned with the discovery of reusable patterns in classes of phenomena. Regularity may be defined as the opposite of randomness, and has given rise to some interesting theories relating to the capturing of regularity

(non-randomness) of long digit-sequences by short descriptions.<sup>22</sup> The study of the abstract theory of regularity could yield practical insights to facilitate the automation of regularity in industrial processes and programming activities.

### Application generators

Application generators strive for reusability in a domain that is narrower than a general-purpose programming language but broader than a specific application.

Application generators such as RPG or Nomad synthesize high-granularity software components over a specialized domain and provide an environment for efficient use of the generated software component. The environment generally contains utilities such as a text editor, a database, and file-handling facilities. Since these utilities are used only with generated software components, they can be custom-designed for this purpose. Application generators can

thus take advantage of two kinds of reusability: (1) reusability of the mechanism for generating software components, and (2) reusability of utilities in providing a service for generated components.

The specialized nature of generated software components is an advantage not only in developing the generating mechanisms but also in developing the environment that supports generated components. The class of generable software components may be specified by a generic program family with richer parameterization facilities than traditional generic programs. Specification of parameter values for a generated software component can be guided by the computer, requiring the user to answer a sequence of questions. The choice of a domain of discourse for an application generator and the design of the generic program generator and parameter interface require a deep understanding of the problem domain.

Application generators for office automation generally have simple user interfaces, such as a questionnaire with questions about office procedures that translate into parameter values of generic software components. Simple interfaces allow users unfamiliar with computers to generate application systems directly, but

are not a necessary requirement for all application generators. Some domains, such as the domain of compilers, may require technical knowledge of programming to generate applications. If the application being generated is important, there is no reason to insist on simple user interfaces. Production-quality systems in

any domain should probably be generated by highly skilled per- expert in both computing and the application domain. Flexibility and evolutionary modifiability of components of an application generator may well be more important than simple user interfaces.

Application generators will make increasing use of expert systems that allow computers to use the knowledge of "experts" in the automatic performance of intelligent tasks in their domain of specialization.<sup>24</sup> For example, office automation systems will make increasing use of "expert" knowledge in automating office procedures. This requires the integration of software engineering techniques for creating application environments with knowledge engineering techniques for creating expert systems. A number of recently formed companies are developing application generators that will facilitate development of expert systems in specific areas from reusable, prepackaged software components.

### Compiler generators

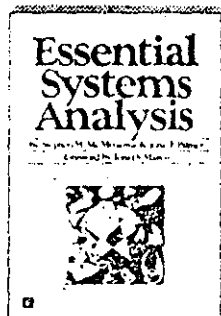
PQCC, the production quality compiler-compiler developed by Wulf, is an interesting example of an application generator in an area where a lot of expert knowledge is available.<sup>23</sup> Compiler-compilers generate programs for a spectrum of programming languages and target machines whose range of variation must be carefully defined. They can make use of the very considerable knowledge of compiler writing accumulated over the last 20 years. Wulf adopts the strategy of breaking down the compiler development process into a large number of subtasks such as parsing, syntactic analysis, and flow analysis. Each is handled by a specialized "software component" with knowledge about that particular task. The program being compiled is transformed from its source language form through a sequence of intermediate representations to its compiled form by a sequence of applications of expert subsystems. The problems of identifying subtasks, encoding expert knowledge about each subtask, providing a mechanism for subtasks to work cooperatively on a larger task, and defining the complete spectrum of applications must initially be handled manually. These tasks could eventually be automated so that not just a single component but a collection of cooperating components can be generated.

NEW FROM YOURDON PRESS!

# Essential Reading for Successful Systems

*Essential Systems Analysis*, by S.M. McMenamin and J.F. Palmer, will increase the accuracy of your specifications and help you speed up the entire systems analysis process. The book introduces methods for modeling *essence* in a new or existing system, focusing on functions that best carry out the system's purpose, without the complications of a specific technology. Using these methods, analysts can distinguish between what a system *has to be* and what one particular implementation *happens to be*. Includes nearly 200 diagrams and examples designed to clarify these new concepts. With *Essential Systems Analysis*, you can add precision to your specifications and save valuable time.

ISBN: 0-917072-30-8; 408 pages; \$28, softcover



SHIPPING: Prices include \$1.25 postage & handling per book. CA, WA, VA & NY state residents, please add applicable sales tax. Orders must be prepaid; or charge it on

VISA or  MASTERCARD.

Card No. \_\_\_\_\_

Signature \_\_\_\_\_

Exp. Date \_\_\_\_\_

I want to ensure accurate specifications. Enclosed is \$29.25 for each copy of McMenamin & Palmer's *Essential Systems Analysis*.

Please send D. King's *Current Practices in Software Development*, a guide to the latest software tools and techniques, at \$29.25 each.

Please send a catalog & future mailings.

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Bus.  or Home  Street: \_\_\_\_\_

City: \_\_\_\_\_ State/Zip: \_\_\_\_\_

Bus.  or Home  Phone: (\_\_\_\_) \_\_\_\_\_

IEEE7.84

Yourdon Press 1133 Ave. of the Americas, New York, N.Y. 10036-6748 800-223-2452 or 212-391-2828 (ext. 3328)

# Part 3: Knowledge Engineering

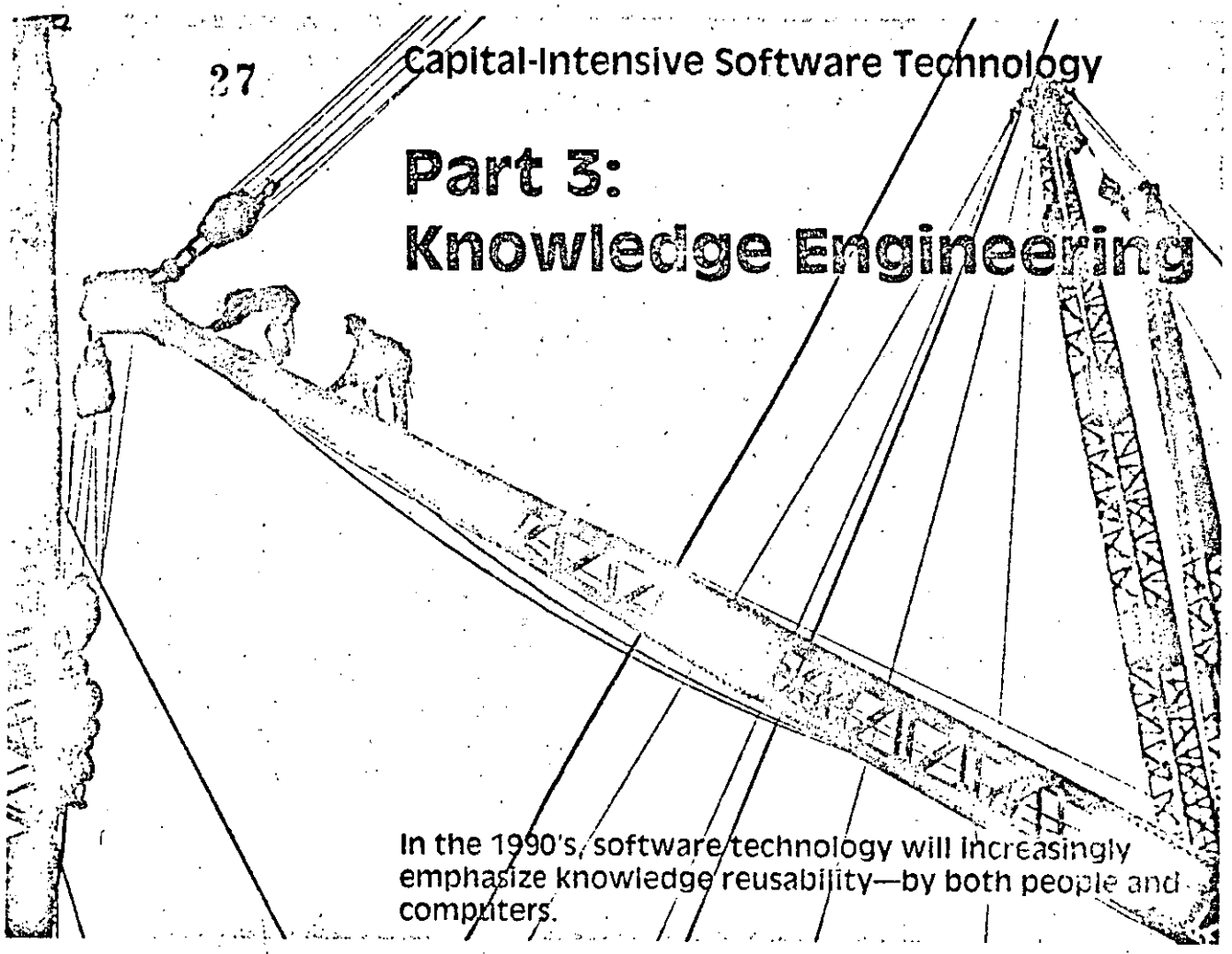


Photo by Lewis W. Hine courtesy of the film America and Lewis Hine (See p. 42)

In the 1990's, software technology will increasingly emphasize knowledge reusability—by both people and computers.

### People-oriented knowledge engineering

The computer revolution will fundamentally amplify man's ability to manage knowledge, just as the industrial revolution fundamentally amplified man's ability to manage physical phenomena.

Knowledge engineering is a body of techniques for managing the complexity of knowledge—just as software engineering is a body of techniques for managing the complexity of software. It is, in that sense, as old as knowledge itself. Euclid's *Elements*, a magnificent piece of knowledge engineering, provided a basis for managing geometrical knowledge, and the classification techniques of Linnaeus are an important example of knowledge engineering in botany and biology. These and other milestones in the development of science are as important for their contributions to the management of knowledge as for their contributions to knowledge

itself. Knowledge engineering is capital-intensive in the sense that reusability is a primary consideration in the development of books, expert systems, and other structures for the management and use of knowledge.

The computer's potential as a tool for knowledge engineering was realized as early as 1945, when Vannevar Bush examined techniques for fundamentally reorganizing knowledge and proposed a device called a memex for the storage, retrieval, and management of knowledge.<sup>25</sup> In the 1960's, Douglas Engelbart proposed a systematic research program on the use of computer technology to augment man's intellectual capabilities.<sup>26</sup> Now, in the 1980's, personal com-

puter technology may allow us to realize their pioneering ideas at an affordable cost.

Feigenbaum, who introduced the term in the context of artificial intelligence, defined "knowledge engineering" as "the art of bringing the tools and principles of artificial intelligence to bear on application problems requiring the knowledge of experts for their solution."<sup>28</sup> This definition views knowledge engineering as the art of representing knowledge so that it can be used by computers to perform intelligent tasks.

The present view is broader. Building knowledge structures to aid human understanding is now seen as the primary objective of knowledge engineering, and intelligent computer problem solving is considered a derivative objective, legitimized by the first objective. The current goal—and motivation—for knowledge engineering is to amplify human intelligence, not to substitute computer intelligence for human intelligence. Its methodology involves educational technology, cognitive science, and human-factors research. The technology of managing the modular presentation of complex knowledge structures has some of the flavor of software engineering, but it must also consider the human factors associated with animation, user interaction, multiple windows, and other techniques for improving man-computer communication.

Personal computers have caused fundamental changes in the way we absorb, manage, and use knowledge.

### The knowledge industry

The production of knowledge as an economic activity, governed in part by market forces of the economy, is a view developed by Machlup in a comprehensive study of the economics and the substance of the knowledge explosion.<sup>27</sup> The importance of knowledge as a product of our economy is shown by the size of the education industry, the growth of the computer industry, and the average age at which people start contributing to the economy. The latter has increased from the early teens during the industrial revolution to over 20 for college graduates and over 25 for doctors and lawyers.

Knowledge is a stock of capital goods, and its production is a capital-intensive activity. The growing importance of the knowledge industry demonstrates that man is becoming an increasingly capital-intensive animal.

New ways of representing and organizing knowledge to exploit interaction, animation, and multiple windows must be developed. More effective techniques for the management of knowledge by humans will complement artificial intelligence techniques for its management by computers, and will result in an environment that integrates human and computational management of knowledge. By the 1990's, knowledge engineering will be as important a subdiscipline of computer sciences as software engineering is today.

### Electronic books

University teachers should initially be appointed to full professorships and suffer a reduction in rank or salary whenever they publish a book or a paper. Then results will be published only if they are sufficiently important to warrant a personal sacrifice.

Restructuring existing knowledge to make it more accessible to humans involves more than putting existing repositories, such as the Library of Congress, on computers and accessing them through information retrieval systems. It involves restructuring existing knowledge so that it can be flexibly presented in different formats for use in different contexts. The technology for such restructuring is not well understood, but its nature can be illustrated by considering recent developments in computerized printing and computer-based learning.

Computers are revolutionizing printing technology to allow rapid, inexpensive reproduction of high-quality text. Word-processing systems increase an author's control over the production, layout, and modification of text. Soon computers will be used not only for writing and printing books but also for reading them. Book-size computers with flat panel displays will make "electronic books" a reality. The greater bandwidth of man-computer interfaces will qualitatively change the nature of man-computer communication; it will make communication of knowl-



edge more effective than today's conventional reading of hard-copy books.

While hard-copy books consist of a linear sequence of pages, materials intended to be read on a computer may have a graph structure with different entry points for readers with different backgrounds. With multiple windows, the reader can pursue several lines of thought simultaneously or view a given object at several levels of detail. The computer can use interactive responses to tailor the graph traversal mode to the individual user's interests and skill level. Each node of the graph structure can include dynamically animated pictures, texts, and programs; for example, the mathematician may animate proof development, while the computer scientist may animate program development and execution.

An electronic book is a family of different hard-copy books that could be obtained by printing out nodes of the graph structure in a particular linear order for particular kinds of students. It is conjectured that flexibility in adapting the pace and order of presentation of information to the student, combined with the power of animation (possibly augmented by voice input and output) can, if properly used, increase enormously the student's capacity to absorb and understand both elementary and advanced knowledge.

### Knowledge support environments

One of the principal objects of research in any department of knowledge is to find the point of view from which the subject appears in its greatest simplicity.

—J. Willard Gibbs

By analogy with the program support environment of software engineering, a computer support system for authors, students, and researchers involved in creating, learning, and using a body of knowledge will be called a *knowledge support environment*. Its desirable features would include systematic support for thinking about problems at multiple levels of

abstraction and for solving problems by divide-and-conquer strategies. Support for both top-down and bottom-up problem-solving strategies would be included. Methods for exploring the design space (solution space) to determine what is possible, and to test the consequences of design decisions by executing partial designs (rapid prototypes), would be supported.

Knowledge support environments would include both facilities for management of knowledge by the human user and facilities for expert systems and interactive man-computer problem solving. Such environments are being developed at Xerox PARC, by manufacturers of personal computers such as the Apollo, Perq, and Sun, and at universities such as Carnegie-Mellon, MIT, and Brown.

### Graph structures of frames

Support environments for authors will receive the same kind of attention in the 1990's that support environments for programmers have received in the 1970's and 1980's.

The knowledge-graph paradigm suggests the organization of knowledge as a graph structure of frames with different entry points and modes of traversal. Knowledge graphs can be used to represent advanced as well as elementary knowledge. For example, a computer-based learning en-

vironment for a programming language such as Ada would include not only introductory material but also a literature of well-documented, prototype "real" programs that examine issues in software methodology. They would be part of a production program support environment. Authors could make use of its tools for production programs, and students could easily switch from the education mode to the production mode.

In working with knowledge graphs, it is conceptually useful to define individual frames as objects of an abstract data type, which will, in the case of programming examples, have a text component, program component, question component, and answer component, plus operations for manipulating each component. A programming language (authoring language) is needed that allows frames in different domains to be declared as different abstract data types. Editors, graph-walking algorithms, answer interpreters, and other tools for creating and using knowledge graphs also need to be defined. Knowledge graphs will require both domain-independent tools that operate on graphs and frames independently of the knowledge domain being considered and domain-dependent tools that know about objects of particular knowledge domains such as programs, forms, and circuits.

### Knowledge graphs

"Knowledge graphs" that can be entered at different points and traversed in different ways impose a modular, interactive discipline on both creators and users of knowledge. They are a basic representation not only for electronic books but also for computer games, such as Adventure, that fascinate by allowing players to explore new graph-structured worlds. Since the hardware technology to support their effective use is only now being developed, we have little experience with building large knowledge graphs, but we can describe their general features.

Knowledge graphs will probably have a domain-independent interconnection structure that facilitates several modes of graph traversal, such as browsing, retrieval, learning, referencing, and authoring. Each node will have a domain-dependent internal structure containing, for example, programs when representing programming knowledge and proofs when representing mathematical knowledge. Graph creators and users will have a domain-independent set of operations for navigating the graph, and domain-dependent operations for manipulating objects in each domain. Zog<sup>29</sup> is an early example of this kind of general-purpose system.

If computer documents consisting of graph structures of frames become a standard mechanism for representing books and computer-based learning materials, then the number of frames produced in the next 30 years will be very large. Thus, capital-intensive aids for reducing the labor and increasing the quality of frame-based documents will be important. Tools can free authors from low-level

tasks and guide them in the higher level tasks of developing insight, understanding, and examples. Work on graphical authoring tools for computer documents has been underway at Brown for several years.<sup>30</sup>

### Dynamic documents

One of the strengths of computer-based knowledge engineering is the ability to switch easily between different levels of abstraction and different views of conceptual objects to gain a more complete understanding of the domain of discourse.

Dynamic documents are intended to be "read" from computers and may combine traditional text with dynamically changing figures and user-interaction facilities. Such documents are particularly effective in presenting information about inherently dynamic objects or processes such as algorithms. The idea of viewing algorithms as processes whose intermediate states are intrinsically interesting rather than as static input-output relations has been explored in depth by Brown and Sedgewick in the context of sorting algorithms.<sup>31</sup>

The key to engineering dynamic documents for algorithms is to find an effective representation of intermediate states that helps the reader gain insight into the execution process. In the case of sorting algorithms, the representation of intermediate states of a partially sorted vector as a graph, which plots the magnitude of each element against its current position in the partially sorted vector, provides remarkable insights into the variety of mechanisms by which sorting algorithms massage elements of a random vector into a sorted vector.

The contrast between the representation of algorithms as input-output relations and as processes with intermediate states is analogous to the contrast between denotational and operational semantics for programming languages. To understand conceptual products such as algorithms or programming languages, it is necessary to understand both their static

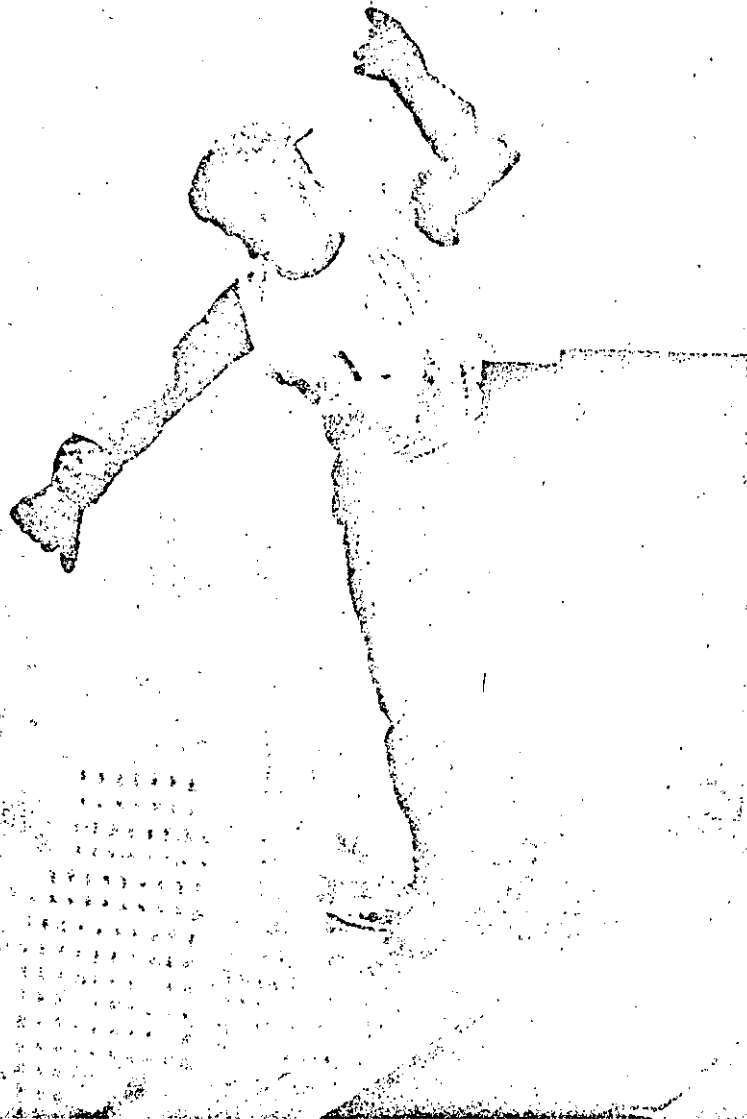


Photo by Lewis W. Hine courtesy of the film *America and Lewis Hine* (See p. 42.)

higher insight. Work or com- derwa characterization as input-output relations or denotations and their dynamic characterization as processes with interesting intermediate states.

Both graph-structured frames and dynamic documents are novel forms of knowledge presentation that are not possible with traditional textbooks. Both are suited for presenting elementary educational material or more advanced material to users of the knowledge or to researchers attempting to extend the frontiers of knowledge. In each case there is an immediate payoff in small documents, constructible with ad hoc techniques, but there is also the promise of much larger payoffs in large documents that extend the user's intellectual reach in significant ways. Clearly, the two styles should be combined, since it is useful for frames to have dynamic components and for dynamic documents to be organized as graph-structured frames.

### Computer authoring technology

Books are large knowledge structures whose problems of management—by both authors and readers—resemble those of large software systems. They are quintessential capital goods, requiring an intense effort to produce, and being reusable by many readers.

Methods for authoring computer textbooks are likely to differ from those for traditional textbooks. Organizing large knowledge domains into graph-structured text modules will require new approaches. The rich visual structure of frames will require authors to think not only about the meaning of words but also about the meaning and communication power of pictures. A disadvantage of the modular approach is that it may violate the subject matter's "natural" continuity, but the advantage is that it requires authors to decompose knowledge systematically into manageable modules.

Authors of the future may have computerized writing assistants that function similarly to pretty printers or programming languages in creating user-friendly representations of knowledge. Writers of textbooks

would no longer have to worry about the surface structure of books at the user interface, but could concentrate on creating a knowledge database of facts, relations, and pictures from which books could be created by "intelligent authoring assistants." The intelligence level of automatic authoring assistants would increase as the technology of converting knowledge databases into user-friendly learning materials became better understood.

Creators and readers of a computer textbook form a social community whose members can communicate directly via a computer message system. Authors can make text available incrementally, receive instantaneous feedback from readers, and rapidly respond to such feedback. Man-computer communication may be used not only for machine display of knowledge but also for communication among its community of creators and users. Such social interaction will permeate all work in knowledge engineering. Computers will affect the sociology of knowledge production by providing a new mode of communication among scholars.

With conventional printing technology, books can be enhanced only by

means of costly new editions. Computer technology, on the other hand, permits continuous incremental enhancement once development of the book has been completed, thereby allowing previously impossible improvements in quality and adaptation to changing requirements. To illustrate the advantages of incremental enhancement, let's compare the life cycle of a program and that of a book. Studies have shown the 80 percent of the effort of supporting a program over its life cycle is in maintenance and enhancement. With conventional printing technology, the only form of "maintenance and enhancement" is printing a second edition, which is time-consuming and expensive. Computer printing technology, by allowing "cheap" incremental maintenance and enhancement, could change the author's role in the life cycle, allowing him a much more active role in both the production and enhancement process.

Knowledge support environments are capital-intensive because they facilitate building and using capital goods. They also encourage capital-intensive practices—initial investments to increase subsequent productivity—by both authors and readers.

**Personal authoring tools**

Knuth has developed a system for writing books about programs that integrates document formatting, program editing, and compiling into a single system, called Web.<sup>32</sup> In this system, a program Weave assembles text and programs into a single readable document that reflects the process by which the program was created. A program Tangle allows programs in the document to be extracted, compiled, and executed.

Knuth's system is recursively illustrated by a 200-page description of Weave and Tangle produced by his system. It presents a remarkably clear, well structured account of those nontrivial programs and illustrates their value for document formatting and word processing. Such systems are evolving into tools that can materially assist authors in the mechanics of authoring, thereby freeing more of their time for the substantive organization and development of ideas.

The Web system is currently restricted to Pascal programs and to be effective requires knowledge of Tex, Web, Pascal, and the use of a systematic programming methodology. It is a personal knowledge-engineering tool tuned for use by specific individuals rather than by a large user community. Knowledge support environments for specific authors will in general start from a general-purpose environment of editing and knowledge-management tools and then include special-purpose tools for supporting the requirements and habits of specific authors. Authors are likely to benefit by investing some of their time in the development of special authoring tools to support their own special needs.

**NOW  
AVAILABLE**

## 8096 SOFTWARE SUPPORT

### COMPLETE 8096 SOFTWARE DEVELOPMENT PACKAGE

Takes A Program From Source To Download To Debug And Checkout

- **COM96** ISBE-96 Host Communications Package
  - Currently supported host systems include IPDS and Series II
  - Support for other host systems is under development
- **ATOP96** Expanded Host Debug Package Includes:
  - Disassembly
  - Single line assembly
  - Informative debug displays
- **XASM96** 8096 Host Cross Assembler

(ISBE-96 is a trademark of Intel Corp.)

**S 8086**

**O 8085**

**F 8051**

**T Z-80**

**W 6502**

**A 6800**

**R 6809**

**E 68000**

### FPAC

#### FLOATING POINT LIBRARIES

- Available for 8086, 8085, 8051, Z-80, 6809
- As low as \$750 (one time fee, no royalties)
- IEEE FORMAT (single and double precision)
- Delivered in source assembly (highly optimized)

FPAC comprises the basic arithmetic operations (add/subtract, multiply, divide), trigonometric functions (COS, SIN, TAN, ATN) logarithms, exponentiation, square root, and data conversions including ASCII to/from floating point and integer to/from floating point.

### MTK

#### MULTI-TASKING KERNEL

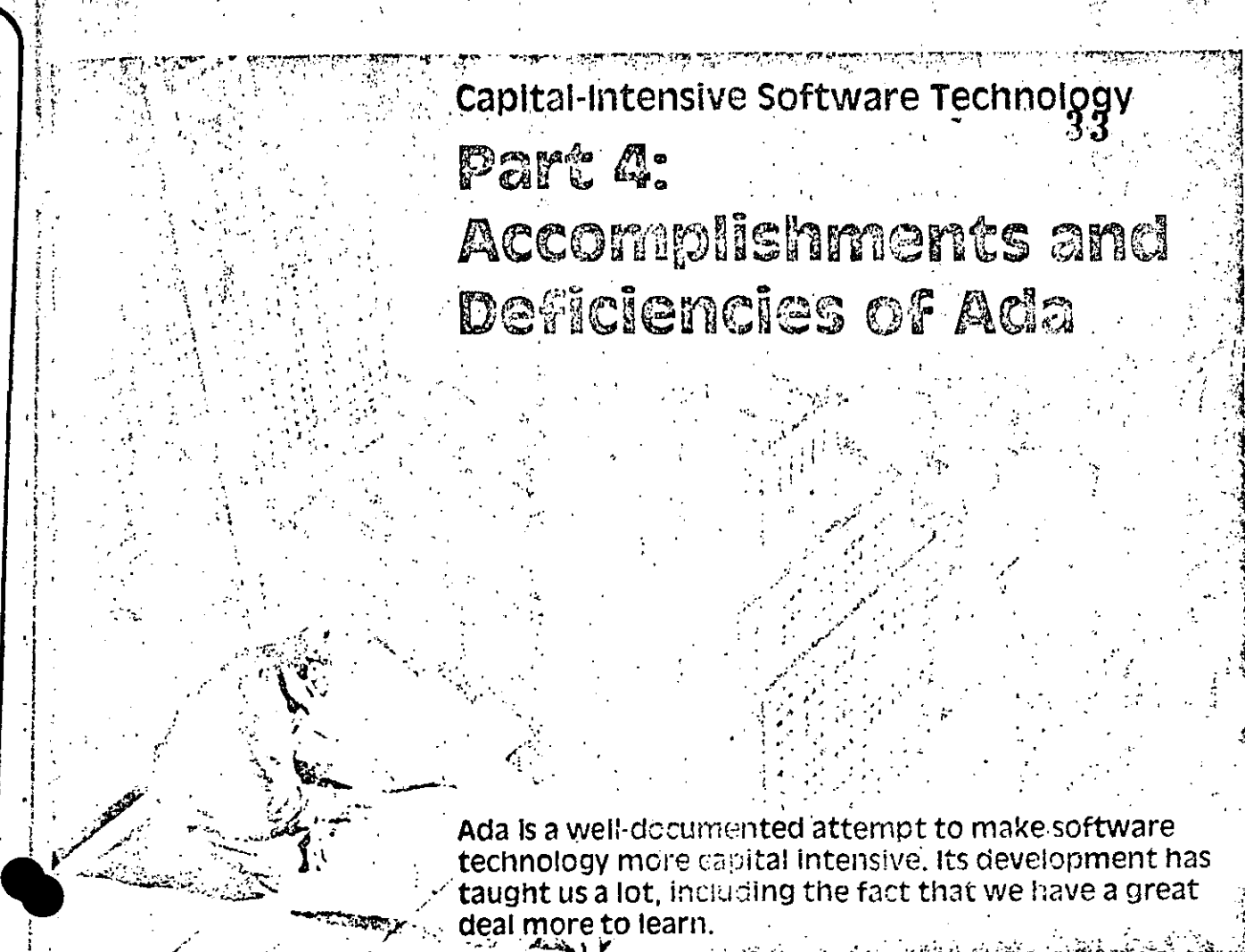
- Real-Time
- Source Assembly
- Easy to Use
- Small (less than 1K)
- 8086, 8085, Z-80, 6502, 6800, 6809, 68000
- Complete Documentation

The Multi-Tasking Kernel including source assembly code for all of the processors listed above is only \$250.

## Capital-Intensive Software Technology

### Part 4:

# Accomplishments and Deficiencies of Ada



Ada is a well-documented attempt to make software technology more capital intensive. Its development has taught us a lot, including the fact that we have a great deal more to learn.

Photo by Lewis W. Hine courtesy of the film America and Lewis Hine (See p. 42.)

### Ada—A case study in capital-intensive software technology

Ada is the second woman mentioned in the Bible—the first after Eve.

Ada and Zillah hear my voice  
Ye wives of Lamech hearken to my speech

--Genesis, Ch. 4, V. 23

Ada was developed to reduce the cost and improve the reliability of software. It is a careful, comprehensive attempt by the world's largest user of software—the US Department of Defense—to develop a capital-intensive framework for software technology.<sup>33</sup> Moreover, the Ada effort is extraordinarily well documented, both in terms of the process by which decisions were made and in terms of documents that describe its requirements and products.

Ada's development began in 1975 with a sequence of programming language requirements, finalized as Steelman in 1978.<sup>34</sup> A language for meeting these requirements was de-

veloped in 1980. However, DoD realized that a language was only a small, if central, component in capital-intensive software technology and thus developed environmental requirements, called Stoneman, during 1978-81. In 1983, Stoneman was supplemented by Methodman, a set of methodology requirements, and by STARS,<sup>35</sup> (Software Technology for Adaptable, Reliable Systems), an intensive study of technology transfer requirements. Thus, the Ada effort has involved four successively broader layers of activity:

- language (reusable by people, computers),

- environment (reusable tools),
- methodology (reusable concepts), and
- technology (education, measurement, integration).

Each of these layers is capital-intensive in the sense of requiring large up-front expenditures to improve later productivity. The improved productivity is achieved by several forms of reusability. Languages are reused by people in writing programs and by computers in compiling and executing them. Environments provide both runtime support and reusable software tools for enhancing programmer productivity. Methodologies determine reusable concepts and techniques for effective problem solving. Technology requirements integrate language, environment, and methodology and address the process of technology transfer.

Ada was designed to support the development of large programs composed of reusable software components. Some of Ada's language features that support such reusability are

(1) A rich variety of program units including subprograms, packages, and tasks. Program units have syntactic interface specifications, which determine the way they may be interconnected in building composite program structures.

#### Reusability in Ada

The Ada effort has spawned a remarkably large number of terms that are near-synonyms of reusability, including the following:

- commonality—reusability of a language by many people;
- portability—reusability of a program or software tool on many computers;
- modularity—reusability of software components in larger applications;
- maintainability—reusability of the unchanged part of a program when a small change is made; and
- evolution—reusability of a system as it evolves in response to changing needs.

(2) Systematic separation between visible syntactic interface specifications and hidden bodies, which allows the programmer to separate concerns about module interconnection from concerns about how the module performs its task.

(3) Strong typing, which imposes constraints on module interconnections and allows consistency between formal parameters of module definitions and actual parameters of module invocations to be enforced at compile time.

(4) Generic program units, which are parameterized templates for generating software components. They allow reusable uniformities of a family of software components to be captured by a single generic definition.

(5) Program libraries with separately compiled reusable program units.

Ada supports a greater variety of software components (abstraction mechanisms) than previous programming languages. This richness is a strength because it increases Ada's expressive power, but it is also a weakness because the different abstraction mechanisms are not well integrated. (See also p. 17.)

The economic benefits of a language like Ada will be determined in part by the technical quality of its language features but in even greater measure by the size of its user community. The real economic payoff comes from standardization that amortizes the cost of development over a large programming community and increases the potential reusability of program modules and tools developed in the language. The objectives of the Ada effort were to achieve a quantum leap forward in capitalization and productivity by combining technical excellence at the level of language features and tools with political and educational mechanisms for rapidly diffusing technical advances over a wide base of users.

The potential benefits of reusability of tools may be illustrated by a recent study of the US Army's software systems. Its 91 major software systems were developed in 43 dif-

ferent languages on 58 computer systems from 29 manufacturers. Each system had its own custom-built support software. In an ideal world of standardized software, the 43 languages could have been replaced by a single language—with great savings in software cost and considerable increases in quality of support. For example, a medium-size system with 10,000 lines of application code that makes use of 100,000 lines of system code would require only 10,000 lines rather than 110,000 lines of new code, and the system would be more reliable because the code could be debugged in a secure environment.

Achieving these savings would require standardization of the environment as well as the language. The Ada approach is to standardize on a Kernel Ada Program Support Environment of nonportable operating system facilities whose interface to the outside world is specified in Ada. The facilities of the KAPSE interface may be used by a much larger set of portable tools that are specified in Ada. Standardization of the environment requires standardization of both the KAPSE interface and the set of tools provided to the user. A standardized environment would allow the 58 systems in the Army example above to be replaced by a single system.

#### Ada—A process or a product?

There is no doubt that we have learned a great deal from the process of developing Ada. But the adoption of Ada as a standard may unduly constrain the evolution of software technology.

The strong economic arguments for adopting Ada as a standard language can be balanced by equally strong arguments for viewing standardization as premature. Ada was developed in a period of rapidly changing software technology. Its requirements for language and environment design represent a static snapshot of an exploding technology at an arbitrary point in its evolution rather

than a stable and mature point of equilibrium appropriate for standardization. Moreover, its initial narrow goals of language standardization have become submerged in much more ambitious and elusive goals of environment standardization. Whereas Ada's goals of language standardization were systematically addressed by the world's top experts in language design, the much more difficult goals of environment standardization, which were not in the original game plan, are being addressed in an ad hoc way by a volunteer committee.<sup>36</sup> Ada's goals have become so diluted and diffuse that they embrace the whole of software technology. Standardization on Ada could have a negative impact on software productivity by channelling resources into Ada that could be more productively used for developing mainstream Ada-independent software technology.

The arguments against standardization on Ada can be summarized as follows:

(1) Ada has involved far more innovation than originally intended because it was developed in a period of transition—from sequential to distributed programming languages and from time-sharing to interactive modes of computer usage. It was innovative in its attempt to integrate the technology of data abstraction and concurrency. It was innovative in its comprehensive attempt to integrate language, environment, and methodology for large evolutionary software systems. Because Ada breaks new ground in so many areas, there are many loose ends in both its language and environment design. This is compounded by the fact that computer technology has evolved so rapidly that the hardware and software assumptions on which Ada was based are almost obsolete.

(2) Ada is a child of 1970's programming language technology. Its block-structure paradigm for language design was dominant in the 1970's but may be replaced by a message-oriented distributed model of computation in the 1980's and 1990's. Its mechanisms for modularity (sub-

programs, packages, tasks, generics, types) are not well integrated with each other and are based on a transient technology.<sup>37</sup> (See also p. 17.)

(3) Ada is a child of 1970's life-cycle technology. It was developed in accordance with the "waterfall" life-cycle model, with a requirements and design phase, and is currently in the middle of its implementation phase. However, this life-cycle model does not accommodate prototyping so that products cannot be tested before being cast in concrete. The prototyping approach advocates throwing away the prototype and starting over to achieve results that are less dependent on early preconceptions. Application of this philosophy to Ada suggests that we throw Ada away and use what we have learned from Ada to develop a new well-integrated language, environment, and methodology.

(4) Ada is a child of 1970's environment technology. Its environmental requirements were developed to be compatible with the timesharing technology of the 1950's and 1960's, before the advent of interactive workstations with high-resolution graphics interfaces. Environment technology has changed even more rapidly than language technology. Although the editing, debugging, and project management tools of Stoneman would be a definite advance over current embedded computing technology, they would be out of date even if they became available in 1985. There is a danger that their widespread introduction in the 1980's could constrain the adoption of the more productive and cheaper interactive environment technology of personal workstations.

(5) Requirements for Ada Program Support Environments are language dependent. The APSE requirement that tools be developed in Ada—and largely for Ada—constrains the scope of Ada environments and makes tool development more expensive. The adverse effects of making the environment Ada-dependent may dominate the consequences of design decisions at the language level. This is a flaw in the overall Ada concept at the

#### Strategic decisions of Ada

The major strategic decisions in the development of Ada included:

(1) language requirements that extrapolate from block-structure languages to encompass data abstraction and concurrency.

(2) choice of Pascal as a "base" for the language,

(3) language-dependent environment requirements that require tools to be written in the language and largely for the language, and

(4) methodology and technology requirements (STARS) strongly coupled to the language.

The first two decisions were reasonable, given the language technology of the 1970's, but deserve to be reexamined in the new circumstances of the 1980's. The third and fourth decisions have unduly constrained environment development and have channeled resources away from the development of productive software environments by forcing simple environment ideas to be implemented by means of a complex language not intended for that purpose.

36 system integration level. It would not be surprising if Ada, just as other very large systems, had its primary problems at this level. The tendency to lavish great care on the internal design of macrocomponents (such as the language) but to exercise weaker control over relations among macrocomponents is a feature that Ada shares with other very large systems.

One of the strongest arguments for Ada is that the mere existence of a standard is more important than the product on which we standardize. According to this argument, the economic benefits of a common language with common subroutine libraries and a common environment will far outweigh any possible differences in quality or approach among candidate programming languages. But these advantages are balanced by potential disadvantages.

(1) Inadequacies of the standard propagate to all its users, causing products that use the standard to be inferior to those that do not.

(2) Once a standard is adopted it may be inflexible, preventing progress. This disadvantage is especially acute in a rapidly changing technology.

(3) The inadequacies of a standard may be propagated to other standards that use it as a basis. For exam-

ple, using a language as a basis for an environment propagates language inadequacies to the environment and constrains the environment design to be dependent on the language.

Ada's standardization may cause problems in each of these areas. Its imperfect software components may give rise to unreliable software systems with components that might be erroneous, particularly for concurrent systems. It standardizes a transitional 1970's language and environment technology that is being rendered obsolete by technological advances, and may stand in the way of a transition to more productive languages and environments. The use of the language as a basis for the environment is running into trouble in part because it is the wrong kind of language, and in part because of our inadequate understanding of environment standardization.

Ada is pioneering new ground in attempting to develop an environment for components with strongly-typed interfaces. In addition, it must overcome the linguistic imperfections of its software components. Language-based environments were successful in the case of Unix (based on C) and Inter-lisp, but their strength derives not from behavioral standardization on toolsets but from

operational standardization on a uniform set of internal system interfaces. In the context of Ada this corresponds to standardization on the KAPSE interface. This has been attempted by a KAPSE Interface Team, which designed a Common Apse Interface Set.<sup>38</sup> But issues in the standardization of internal system interfaces are not well understood, particularly in the presence of strong typing, and standardization may well be premature. Developing Ada-based interface standards may not be as productive an approach as starting from a demonstrably proven base, such as Unix, and extrapolating from this base—just as Ada extrapolated from Pascal.

Reusable software technology requires standardization not only of languages and environments, but also of major software subsystems like communication, database, and workstation subsystems. Little is being done to coordinate language standardization with subsystem standardization. There is an opportunity in the development of large DoD software systems, such as WIS, with a budget in excess of \$30 billion over 15 years, to develop communication, database, and workstation subsystems that could become a de facto standard for other very large applications. Subsystem standardization could provide dividends in productivity that dominate those of language standardization.

The problem of standards in software technology is complex because of the strong interaction among its diverse elements. Standards are needed not only for languages and environments, but also for software acquisition, life-cycle methodology, documentation, and a whole range of other technological elements. Standardization on one element, such as language, means that it must remain fixed while other elements evolve. This can be an advantage if we are confident that the standard is appropriate and stable, but can seriously constrain and distort overall evolution of the technology if the standard is inappropriate.

### America and Lewis Hine

Between 1904 and 1926, 14 million men, women, and children poured through Ellis Island to become part of the labor force that would build industrial America. Photographer Lewis W. Hine (1874-1940) was present at this great drama. For the next 35 years, in over 15,000 photographs, Hine followed the story of these newcomers as they lived and labored to build America. Hine's dedication to documenting the story of the immigrants led him to become the staff photographer of the National Child Labor Committee, and his passionate photographs of children at work were instrumental in the crusade to pass child labor legislation.

In the 1920's, Hine undertook a series of "work portraits" in which he emphasized the courage and skill of workers, who were to him still primary in what was then hailed as the "machine age." In 1930, at the age of 56, he became the official photographer of the Empire State Building.

Hine's extraordinary photographs, nine of which are reproduced within this article, are now recognized as priceless treasures of our national heritage. An hour-long documentary on his life and work, produced by Daedulus Productions, Inc., is scheduled for prime-time broadcast nationwide on PBS this fall. The film, *America and Lewis Hine*, is more than a portrait of an individual artist—it is a compelling testament to the strength of those who built America.



## Conclusion

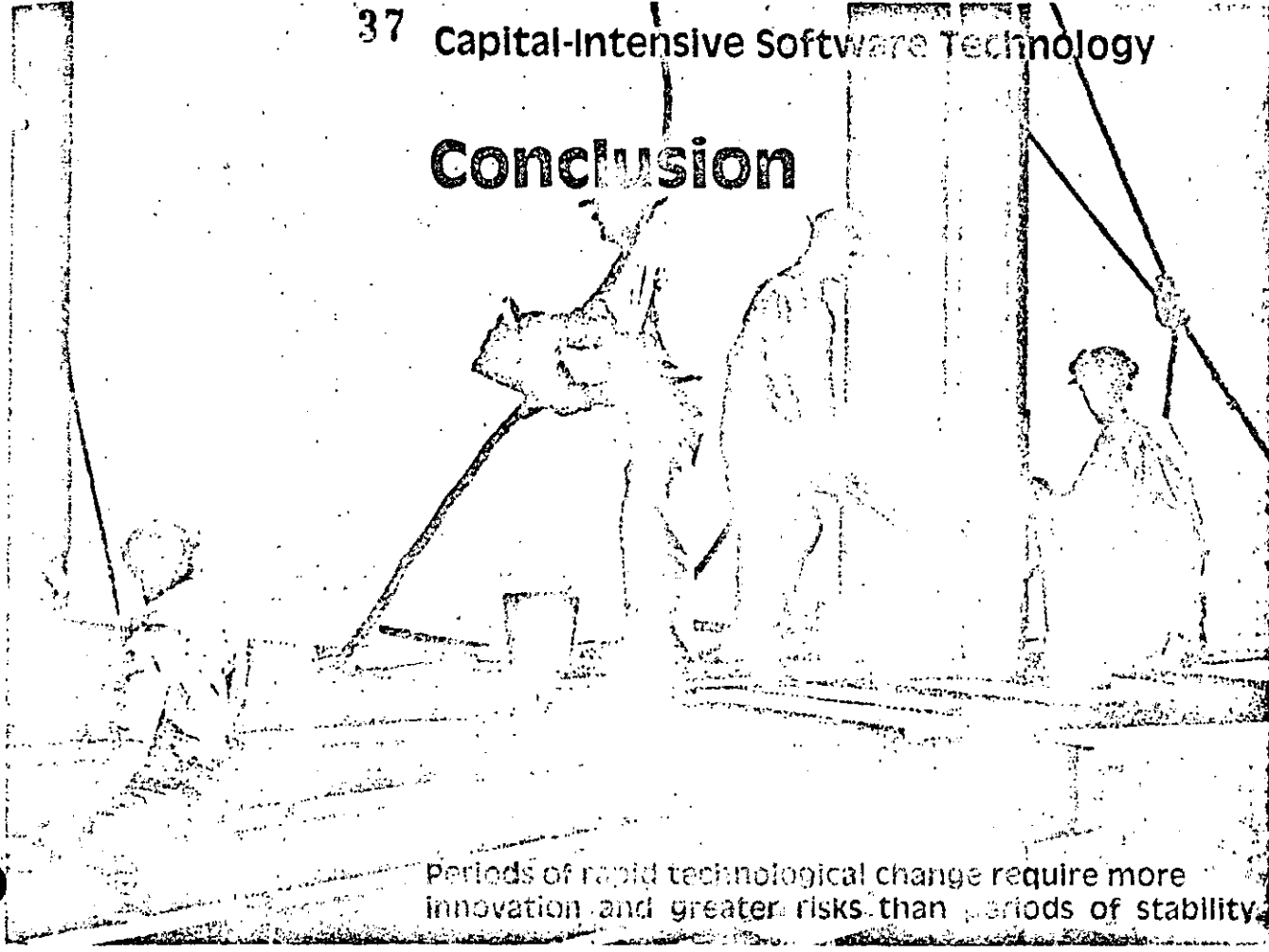


Photo by Lewis W. Hine courtesy of the film America and Lewis Hine (See p. 42.)

Periods of rapid technological change require more innovation and greater risks than periods of stability.

Just as the Eskimo has many different words for snow, we have many words for reusability. A plausible conclusion is that reusability of the resources we create is as important in our lives as snow is in the life of the Eskimo.

Reusability has provided us with a single metric for examining a variety of software activities—software components, programming in the large, knowledge engineering, and Ada.

In the area of software components, the shift from sequential to concurrent models of computation has opened up new dimensions in language and environment design. Sequential programming languages such as Pascal and Ada were based on the block-structure paradigm. Concurrent programming languages are still in the pre-paradigm stage of development, but the plug-and-socket distributed-sequential-processes paradigm appears attractive on the grounds of both simplicity and logical expressiveness. The programming language NIL embodies the new paradigm. Its com-

ponents are free from the textual bonds of block structure, can perform autonomous concurrent computations, can be linked and reconfigured dynamically while they are executing, and are designed for systems that may evolve during program execution as well as during program development. NIL's greater autonomy, on the other hand, carries with it responsibilities to efficiently implement data protection, communication, and recovery.

In the area of programming in the large, there is intense debate about a paradigm shift from the waterfall life-cycle model to an interactive model that uses the computer as an integral part of the problem-solving process. The new paradigm includes a shift from behavioral to operational specification, with emphasis on rapid (car-

ly) prototyping. It aims to provide automatic transformations from high-level operational specifications to efficient implementations. This paradigm has yet to prove itself. But there is no doubt that the availability of cheap computing power as an almost free resource in interactive problem solving will have a profound impact on the problem-solving process.

Knowledge engineering has been redefined to emphasize providing aids for augmenting rather than replacing human intelligence. Knowledge management is less ambitious than artificial intelligence, but can provide a framework within which both computers and humans function more intelligently. Knowledge engineering (in the new sense) subsumes software engineering, since software engineering is simply knowledge engineering applied to the specialized domain of software. But software engineering needs domain-independent knowledge-management tools for tasks like documentation, authoring, and library management. Program support environments for software develop-

ment are therefore dependent on good knowledge-support environments. Capital-intensive software technology requires a knowledge-support environment for programmers to be productive, and in turn provides the technology that allows good knowledge-support environments to evolve.

Our discussion of Ada took a broad view, examining the capital-intensive nature of the language, environment, methodology, and technology. We indicated that Ada involved far more innovation than originally intended, in part because it had the misfortune of being developed during a period of technological transition from sequential to concurrent software components and from batch to interactive environments. Its careful attempt at standardization might have succeeded had the technology been more stable, but instead the effort has resulted in a transitional product that mirrors the transitional language and environment technology which spawned its development.

The US Department of Defense should hitch its wagon to STARS of the future rather than the past. It should build on the ideas of Ada but bypass Ada as a product. Perhaps it could support a new competition to specify language and environment designs for the technology of the late 1980's and the 1990's. New requirements would probably start with environment and interface requirements at the KAPSE and Unix levels, continue with communication requirements for software components and libraries, and add language requirements once interface and communication requirements had been agreed upon. Greater emphasis would be placed on the standardization of communication interfaces and less on the standardization of computation primitives.

Reopening the language and environment standardization issue will delay adoption of a standard. But this delay may well occur anyway because of the inadequacies of Ada implementations and the pressures of new technology. A reexamination of language and environment standards could be sponsored by the DoD Software Engineering Institute proposed as a central feature of the STARS program.

Many of us who work in the field of software technology feel that the 1980's are more exciting than the 1970's and that the 1990's may prove to be even more so. Living in a period of rapid technological change provides both an opportunity and a responsibility for shaping the future. It requires us to be more innovative and to take greater risks than in a period of greater stability. But worthwhile progress can be achieved only by taking some risks, making some hard decisions, and investing in the future.

The French maxim "reculer pour mieux sauter" (draw back to better jump) echoes the capital-intensive sentiment of giving up present profit for future productivity, and suggests that drawing back from a commitment strengthens our ability to face the future.

Photo by Lewis W. Hine courtesy of the film *America and Lewis Hine* (See p. 42.)



Ackn  
This  
om  
ed  
distrib  
Brian  
and the  
ing co  
Part  
vided  
tract N  
No. 47

Refs

Refer

1. S
- B
2. R
- E
- S
- L
3. C
4. I
- 5.
- 6.
- 7.
- 8.
- 9.

Ref

- 10.
- 11.
- 12.
- 13

## Acknowledgments

This article owes a great deal to Rob Strom and Shaula Yemini, who introduced me to the world of NIL and distributed processing. Dennis Allison, Dan Dalio, Rob Rubin, Bruce Shriver, and the students of my software engineering course contributed to its debugging. Partial support for this work was provided by ONR and DARPA under Contract N00014-83-K-0146 and ARPA Order No. 4786, and by IBM Yorktown Heights.

## References

### References for Part 1

1. S. R. Bourne, "The UNIX Shell," *Bell System Technical J.* July-Aug. 1978.
2. R. Strom and S. Yemini, "NIL, An Integrated Language and System for Distributed Programming," *Proc. Sigplan 83 Symp. Programming Language Issues in Software Systems*, June 1983.
3. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, Oct. 1974.
4. B. Liskov and R. Scheifler, "Guardians and Actions: Robust Support for Distributed Programs," *Proc. POPL Conf.*, Jan. 1982.
5. P. Wegner, "On the Unification of Data and Program Abstraction in Ada," *Proc. Principles of Programming Conf.*, Jan. 1983.
6. R. Strom and S. Yemini, "Optimistic Recovery: An Asynchronous Approach to Fault Tolerance in Distributed Systems," *Proc. FTCS-14*, June 1984.
7. Special Issue on Smalltalk, *Byte*, Aug. 1981.
8. D. Weinreb and D. Moon, *Lisp Machine Manual*, MIT, 1981.
9. C. W. Bachman, "The Programmer as Navigator," *Comm. ACM*, Nov. 1973.

### References for Part 2

10. E. Feigenbaum and P. McCorduck, *Fifth Generation Computers*, Addison-Wesley, Reading, Mass., 1983.
11. P. Zave, "The Operational Versus the Conventional Approach to Life Cycle Development," *Comm. ACM*, Feb. 1984.
12. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
13. R. Balzer, T. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, Vol. 16, No. 11, Nov. 1983, pp. 39-45.

14. V. R. Basili and A. J. Turner, "Iterative Enhancement, A Practical Technique for Software Development," *IEEE Trans. Software Engineering*, Dec. 1975.
15. A. J. Toynbee, *The Study of History*, Oxford University Press, 1947.
16. J. G. March and H. A. Simon, *Organizations*, Wiley, New York, 1958.
17. E. H. Schein et al., *Organizational Development*, Addison-Wesley, Reading, Mass., 1973.
18. J. Holland, *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press, Ann Arbor, 1975.
19. C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, June 1977.
20. R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science No. 92*, Springer-Verlag, New York, 1980.
21. W. Scherlis, "Specialization," *Proc. POPL Conf.*, Jan. 1981.
22. G. J. Chaitin, "Randomness and Mathematical Proof," *Scientific American*, 1975.
23. W. A. Wulf, "PQCC: A Machine-Relative Compiler Technology," Report No. CMU-CS-80-144, Carnegie-Mellon Univ., Pittsburgh, Pa. 1980.

### References for Part 3

24. M. Stefik et al., "The Organization of Expert Systems—A Tutorial, Artificial Intelligence," Mar. 1982.
25. V. Bush, "As We May Think," *Atlantic Monthly*, July 1945.
26. "A Conceptual Framework for the Augmentation of Man's Intellect," in *Vistas in Information Handling*, Vol. 1. Howerton and Weeks, eds., Spartan, 1963.
27. F. Machlup, *Knowledge, Its Creation, Distribution, and Economic Significance*, Princeton University Press, Princeton, N. J., 1980.
28. E. A. Feigenbaum, "Case Studies in Knowledge Engineering," *Proc. Fifth Int'l Conf. Artificial Intelligence*, Aug. 1977.
29. G. Robertson, D. McCracken, and A. Newell, "The Zog Approach to Man-Machine Communication," *Int'l J. Man-Machine Studies*, 1981.
30. S. Feiner, S. Nagy, and A. van Dam, "An Experimental System for Creating and Presenting Interactive Graphical Documents," *ACM Trans. Graphics*, Jan. 1982.

31. M. Brown and R. Sedgewick, "Techniques for Algorithm Animation," technical report CS-84-02, Brown University, Providence, R. I., Jan. 1984.
32. D. E. Knuth, "The WEB System for Program Documentation," Stanford University Report, Stanford, Calif., 1982.

### References for Part 4

33. *Ada Reference Manual*, US Department of Defense, ANSI/MIS-STD 1815, Jan. 1983.
34. "Requirements for Higher Order Computer Programming Languages—STEELMAN," in *Tutorial on Programming Language Design*, A. I. Wasserman, ed., Computer Society Press, Los Alamitos, Calif., 1980, pp. 298-315.
35. Special Issue on the DoD STARS Program, *Computer*, Vol. 16, No. 11, Nov. 1983.
36. G. Fisher, Chairman's Letter, *Ada Letters*, Mar.-Apr. 1984.
37. R. Strom, P. Wegner and S. Yemini, "Ada is Too Big," draft report, IBM, Yorktown Heights, Apr. 1984.
38. Common Apse Interface Set, Version 1.0, Ada Joint Program Office, Aug. 1983.



Peter Wegner, a professor of computer science at Brown University, has taught at the London School of Economics, Penn State, and Cornell. His current interests include programming languages for distributed computation and workstation environments for the 1990's; his recent work includes papers on abstraction mechanisms and concurrency in programming languages, Ada education and technology transfer, and paradigms of computer science. He has been associated with Ada since 1977 when he coauthored the recommendation that a common language based on Pascal, Algol 68, or PL/I be developed. Wegner's six books include *Programming in Ada* (Prentice-Hall, 1980) and *Research Directions in Software Engineering* (MIT Press, 1979). He was educated in England at London and Cambridge University.

The author's address is Brown University, Dept. of Computer Science, Box 1910, Providence, RI 02912.



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

MAJOR ISSUES IN SOFTWARE ENGINEERING PROJECT MANAGEMENT

DICIEMBRE, 1984



# Major Issues in Software Engineering Project Management

RICHARD H. THAYER, MEMBER, IEEE, ARTHUR B. PYSTER, MEMBER, IEEE,  
AND ROGER C. WOOD, MEMBER, IEEE

**Abstract**—Software engineering project management (SEPM) has been the focus of much recent attention because of the enormous penalties incurred during software development and maintenance resulting from poor management. To date there has been no comprehensive study performed to determine the most significant problems of SEPM, their relative importance, or the research directions necessary to solve them. We conducted a major survey of individuals from all areas of the computer field to determine the general consensus on SEPM problems. Twenty hypothesized problems were submitted to several hundred individuals for their opinions. The 294 respondents validated most of these propositions. None of the propositions was rejected by the respondents as unimportant. A number of research directions were indicated by the respondents which, if followed, the respondents believed would lead to solutions for these problems.

**Index Terms**—Project management, software engineering, survey, university curriculum.

## I. INTRODUCTION

**N**EARLY every software engineering development project is plagued with numerous problems leading to late delivery, cost overruns, and sometimes, unsatisfied customers. Often these problems are technical, but just as often, the software engineering development problems are managerial. How often have we personally heard or read that a project was late (or overbudget, or reduced in scope, or terminated early, or did not satisfy the user, . . .) because:

- programmers did not tell the truth (or did not know the truth) about the status of their programs, or
- management unreasonably reduced the delivery time of (or budget of, or withheld necessary resources from) the project, or
- top management did not allow sufficient time for front end planning, or
- the true status of the project was never known, or
- programmer productivity was lower than planned, or
- the customer did not know what he wanted (or changed his requirements), or
- government standards for requirement specifications (or procurement policies) were not suitable for software, or . . .

Although the technological and managerial aspects of software engineering were both recognized about the same time

Manuscript received May 22, 1979; revised February 5, 1980. Any opinions expressed herein are those of the authors and do not necessarily reflect the opinions of the U.S. Air Force.

R. H. Thayer was with the Sacramento Air Logistics Center, Air Force Logistics Command, McClellan Air Force Base, CA 95652. He is now with the Department of Computer Science, California State University, Sacramento, CA 95819.

A. B. Pyster and R. C. Wood are with the Department of Computer Science, University of California, Santa Barbara, CA 93106.

[25], improvements and developments in management have not kept pace with advances in the technology of software development. A large number of articles addressing such topics as better coding style ("structured programming"), testing, formal verification, language design for more reliable coding, diagnostic compilers, and so forth, have appeared in the literature (e.g., in the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Proceedings of the International Conferences on Software Engineering*, *Proceedings of the ACM Conferences on the Principles of Programming Languages*, . . .). Although the technology of software engineering as a well-defined discipline is relatively new, software engineers have progressed to the point where many major issues relevant to the technology of software production have been identified and considerable progress in addressing these issues has been made [16], [17]. Practical working tools to support improved software production are commonly available, and their design and generation have become a recognized topic for university instruction [2].

Software engineering project management (SEPM) has not enjoyed the same progress. While it might be argued that SEPM has been defined, it is far from a recognized discipline. Software developers who have demonstrated competence as developers and programmers have been elevated to project managers without the benefit of education or training. The major issues and problems of SEPM have not even been agreed on by the computing community as a whole, and, consequently, priorities for addressing them have not been widely established. Research in SEPM has been scant. As Commander Cooper reported in the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING Special Issue on Project Management (July 1978):

"Although the need is apparent, there appears to be precious little innovative activity in the area of software management. Perhaps this is so because computer scientists believe that management per se is not their business, and the management professionals assume that it is the computer scientists responsibility."

Richard Merwin stated in the same issue:

"Programming disciplines, such as top-down design, use of standard high level languages, and program library support systems all contribute to the production of reliable software on time, within budgets. . . . What is still missing is the overall management fabric which allows the senior project manager to understand and lead major data processing development efforts."

Some data were collected about the extent to which univer-

sities teach SEPM. It revealed that only a handful of prominent universities surveyed had any courses exclusively on SEPM. On the other hand, most of these universities offered at least one course on the technological issues of software engineering. More details on this study are presented in Section VI.

Before we can agree on the progress accomplished in SEPM, we must establish the yardstick for measuring that progress. The first step must be to identify the major issues and problems which face project managers. This paper will attempt such an identification based on a survey of 294 individuals. In addition, it will try to indicate the relative importance of these issues and problems.

## II. THE MAJOR ISSUES

The *major* or *key issues* of an activity, technology, or task are those portions of it which are:

- critical to the success or excellence of the larger activity, technology, or task, *and*
- have an existing or potential weakness which may significantly detract from the success or excellence of the activity, technology, or task, or in a worst case, cause failure.

The major issues of SEPM must be identified in order that:

- real problems can be separated from pseudoproblems enabling management attention to be properly focused,
- the university/education system can properly apply its resources in the training of software managers as well as engineers,
- a greater level of abstraction can be identified to provide a broader basis for understanding the management of a software development project.

The first step taken to define the major issues of SEPM was to review the literature for software engineering problems since 1974 (plus several early classical documents) concentrating on secondary sources in order to take advantage of generalizations already made. By using the software engineering delivery and success model shown in Fig. 1, we hypothesized which of these problems can most affect the success of software delivery. These, we believed, were the major issues.

Each software engineering issue was then evaluated from the viewpoint of the project manager, leaning towards a manager who does not have a great deal of managerial experience and is seeking a means to ensure delivery of a successful project. These issues were then reworded as *problems* as seen by the project manager and compared to the classic management model of planning, organizing, staffing, directing, and controlling to ensure that every area was covered. By far the dominant two activities are planning and controlling, which together account for 80 percent of the issues, with planning alone involving ten issues. The resulting 20 major issues of SEPM are shown in Table I, along with their respective source references and a short title for later reference.

## III. VALIDATING THE ISSUES

Based upon the criteria explained in the last section, the 20 problem statements shown in Table I were assembled. The

2

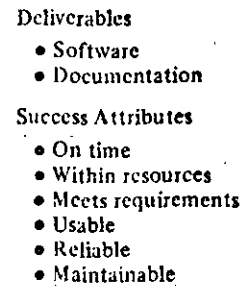


Fig. 1. Software development delivery and success model.

next step was to attempt to validate that these 20 "problems" are truly problems for project managers in the field. Two reasonable validation methods in this context are to:

- perform a detailed case study of a number of projects, observing the problems encountered by project managers,
- survey various individuals for their opinions on the nature of managerial problems in software development projects.

In fact, both methods were tried. The results of the case study will be reported in a forthcoming paper. The second approach, surveying, is the method reported here for validating the problems which is reported here.

The survey, conducted between Autumn 1977 and Spring 1979, obtained information from industrial, governmental, and university leaders on the major problems of SEPM. The categories of participants desired and eventually obtained is shown in Fig. 2. Experienced and knowledgeable data processing managers and other personnel from the data processing area were sent a copy of the 20 issues and asked to state their opinion as to whether they felt each of the hypothesized problems was "critical," "important," "not important," "not a problem," or, lastly, "incorrectly stated." In addition, the respondents were asked whether they viewed a problem as being primarily managerial, technical, both, or neither. The respondents were also asked to complete two more questions about each problem. They were asked whether the solution to the problem was obtainable through improvements in management, technology, both, or neither. Finally, they were asked to state in English prose, how they would (or did) solve this problem. The last question helped ensure that the participants thought through their answers with more than a casual effort. A sample of one survey question is contained in Fig. 3.

The participants were obtained from published names and addresses available to the general computing community and by distributing survey copies to people attending computer-oriented conferences. The survey was mailed to highly visible individuals in government and private computer science, particularly members so the IEEE Computer Society, ACM, and AFIPS. In addition, the survey was distributed at a number of conference sessions in which the emphasis was on project management. In addition, as word of the survey spread, we received dozens of requests for survey copies as well as recommendations for other candidate participants.

To further increase the pool of potential respondents, we examined several widely read computer journals for project management articles and software engineering articles that could be related to project management, and mailed a survey

TABLE I  
 3. TWENTY HYPOTHESIZED PROBLEMS IN SOFTWARE ENGINEERING PROJECT MANAGEMENT

<u>Planning Problems</u>	
<i>Problem 1 (Plan Requirement):</i> Requirement specifications are frequently incomplete, ambiguous, inconsistent, and/or unmeasurable [31], [30], [32], [3], [15], [12], [5], [7], [10], [25], [13], [9], [11].	<i>Problem 12 (Organization Accountability):</i> The accountability structure in many software engineering projects is poor, leaving some question as to who is responsible for various project functions [15], [29], [5], [13].
<i>Problem 2 (Plan Success):</i> Success criteria for a software development are frequently inappropriate which results in poor "quality" delivered software, i.e., not maintainable, unreliable, difficult to use, relatively undocumented, etc. [15], [32], [3], [22], [5], [10], [14], [28], [9], [11].	<u>Staffing Problems</u>
<i>Problem 3 (Plan Project):</i> Planning for software engineering projects is generally poor [6], [15], [22], [24], [12], [5], [10], [28], [13].	<i>Problem 13 (Staff Project Manager):</i> Procedures and techniques for the selection of project managers are poor [15], [12], [5], [10], [13].
<i>Problem 4 (Plan Cost):</i> The ability to estimate accurately the resources required to accomplish a software development is poor [1], [32], [8], [15], [29], [5], [10], [28], [11], [18].	<u>Directing Problems</u>
<i>Problem 5 (Plan Schedule):</i> The ability to estimate accurately the delivery time on a software development is poor [1], [30], [32], [8], [15], [34], [5], [10], [28], [13], [18].	<i>Problem 14 (Direct Techniques):</i> Decision rules for use in selecting the correct management techniques for software engineering project management are not available [10].
<i>Problem 6 (Plan Design):</i> Decision rules for use in selecting the correct software design techniques, equipment, and aids to be used in designing software in a software engineering project are not available [19], [15], [12], [5], [10], [28], [11].	<u>Controlling Problems</u>
<i>Problem 7 (Plan Test):</i> Decision rules for use in selecting the correct procedures, strategies, and tools to be used in testing software developed in a software engineering project [3], [12], [10], [28], [35].	<i>Project 15 (Control Visibility):</i> Procedures, techniques, strategies, and aids that will provide visibility of progress (not just resources used) to the project manager are not available [19], [31], [8], [15], [29], [5], [12].
<i>Problem 8 (Plan Maintainable):</i> Procedures, techniques, and strategies for designing maintainable software are not available [12], [7], [11].	<i>Problem 16 (Control Reliability):</i> Measurements or indexes of reliability that can be used as an element of software design are not available and there is no way to predict software failure, i.e., there is no practical way to show the delivered software meets a given reliability criteria [3], [21], [22], [12], [10], [20], [28], [9].
<i>Problem 9 (Plan Warranty):</i> Methods to guarantee or warrant that the delivered software will "work" for the user are not available [30].	<i>Problem 17 (Control Maintainability):</i> Measurements or indexes of maintainability that can be used as an element of software design are not available, i.e., there is no practical way to show that a given program is more maintainable than another [28], [9].
<i>Problem 10 (Plan Control):</i> Procedures, methods, and techniques for designing a project control system that will enable project managers to successfully control their project are not readily available [19], [15], [5], [28], [9], [11].	<i>Problem 18 (Control Goodness):</i> Measurements or indexes of "goodness" of code that can be used as an element of software design are not available; i.e., there is no practical way to show that one program is better than another [32], [26], [33].
<u>Organizing Problems</u>	<i>Problem 19 (Control Programmers):</i> Standards and techniques for measuring the quality of performance and the quantity of production expected from programmers and data processing analysts are not available [1], [31], [32], [15], [29], [34], [28], [11].
<i>Problem 11 (Organization Type):</i> Decision rules for selecting the proper organizational structure, e.g., project, matrix, function, are not available.	<i>Problem 20 (Control Tracing):</i> Techniques and aids that will provide an acceptable means of tracing a software development from requirements to completed code are not generally available [3], [28], [5].

copy to the articles' authors. Finally, the survey was distributed to a group of professional programmers and to one graduate class in computer science at the University of California at Santa Barbara. These last two distributions account for the majority of the project individuals (as opposed to managers) and students who completed the survey.

As a twist on the original surveying method, we also sent a modified questionnaire to most major universities in the country which offer computer science degrees to determine to what degree SEPM was being taught. Twenty-seven responses were received, revealing that very little on SEPM is currently incorporated into the regular computer science programs at either undergraduate or graduate levels. Section VI contains the survey details.

#### IV. THE SURVEY RESULTS

Our primary goal was to uncover whatever consensus exists among those involved in software development on the major

problems in SEPM. It was not our intent to determine why the participants answered as they did, nor to analyze the relationships between the various propositions or their parts. We feel that because the problem statements are short, and subject to different interpretations, the reader should not place too much significance on the specific percentages tabulated for individual problems. Therefore, to determine whether one of the hypothesized problems was truly a problem or not, a 30/40/30 rule was adopted. If fewer than 30 percent of the respondents felt that a proposed problem was at least "important," then the hypothesis would be rejected. On the other hand, if at least 70 percent felt that the issue was either critical or important, then the hypothesis was accepted and that issue categorized as a major problem. Finally, if at least 30 percent but less than 70 percent of the respondents felt that the problem was important (the middle 40 percent), then the hypothesis could not be conclusively accepted or rejected. No more refined ranking of the problems was done for this study. The



- Technical leaders in computer science who are/have either:
  - a position of influence in their company,
  - highly visible authors and/or speakers on computer science or project management
- Project managers
- R&D personnel
- Educators (university, education institute, ...)
- Other personnel interested in project management

Fig. 2. Desired participants.

1. PROBLEM--Requirement specifications are frequently incomplete, ambiguous, inconsistent, and/or unmeasurable.
- a. This problem is: critical ( ) ... important ( ) ... not important ( ) ... no problem ( ) ... incorrectly stated ( )
  - b. This is a problem in: management ( ) ... technology ( ) ... both ( ) ... neither ( ) ... not a problem ( )
  - c. This problem can be solved through improvements in: management ( ) ... technology ( ) ... both ( ) ... neither ( ) ... not a problem ( )
  - d. How would (did) you solve this problem? \_\_\_\_\_

Fig. 3. Sample survey question.

choice of the 30/40/30 figures is somewhat arbitrary, but does, we believe, reflect the notion of "consensus" fairly well. We have provided the percentage figures in the tables presented in this section so that if the reader is unhappy with our classification scheme, the information to construct a new scheme is available.

The 30/40/30 rule can also be applied to parts B and C of the questionnaire. Parts B and C, which refer to the problem type and solution type respectively, were evaluated to determine whether the problems and solutions are managerial, technical, both, or neither. If managerial has a weighted average of at least 70 percent, then the problem (solution) type is considered to be managerial; if fewer than 30 percent felt that the proposition is a managerial problem (solution), then the proposition is considered to be technical; if the split between managerial and technical falls in the center 40 percent, then we conclude that the problem (solution) has both a strong managerial and a strong technical character and cannot be characterized as either managerial or technical alone. The weighted managerial average is defined to be

$$\frac{\text{NR MGT} + 0.5 (\text{NR BOTH})}{\text{NR MGT} + \text{NR TECH} + \text{NR BOTH}}$$

where "NR MGT" is the number of surveyees that answered part B (C) with "management," "NR TECH" is that number that answered "technical," and "NR BOTH" is the number that answered "both."

Two hundred and ninety-four survey replies were received in all including those mailed to specific individuals and those returned from handouts at conferences and the classroom. Approximately 25 percent of those specifically addressed to an individual were returned. Respondents ranged from high ranking decision makers in computer science, senior corporate

4

TABLE II  
ATTRIBUTES OF PARTICIPANTS

GROUP	ATTRIBUTE	PERCENTAGE OF RESPONDENT
Job or position (only one applies)		
	line manager	10
	project manager	13
	individual developer	21
	senior staff, policy supervisor, software	4
	corporate staff, software	12
	supervisor, non-software	6
	anonymous	1
	university teaching	-
	consultant	28
	consultant	3
General (more than one applies)		
	R&D oriented	23
	educators	26
	faculty CS Dept	20
	student	7
	programmer/software analyst	21
	engineer/functional analyst	9
	quality assurance/technical director	2
	manager/supervisor	45
	government employee	24
	PhD	17
	consultant	10
	establishes general software development policy	8
	national author/speaker	24
	affiliated with CS professional organization	36
	affiliated with aerospace professional organization	14
	American/Canadian influence	96
	European influence	6
	Far East influence	1
Employer or Firm (only one applies)		
	manufacturer of computer hardware	15
	manufacturer of other than computer hardware	12
	software house	3
	engineering services and technical support organization	8
	government	23
	university/R&D laboratory	31
	computer service bureau	1
	consulting firm	4
	financial institute	1
	medical/legal service	1
	utilities	1

and DoD officials, to legendary figures in the computing field, and project managers and programmer/analysts on a software project. Many respondents publish regularly in leading technical journals and are highly sought-after speakers at conferences in the the U.S. and abroad. The authors of well-known textbooks and leading researchers both within the university system and at R&D laboratories were well represented. Although the survey participants can be considered a cross section of the computing community (see Table II), by design the emphasis was to obtain those individuals who through visibility position sway the opinions of many others working in software engineering or other aspects of data processing management. Of course, the individual names of the respondents are protected by a promise of confidentiality.

One of the more interesting possibilities in analyzing the

TABLE III  
CATEGORIES OF PARTICIPANTS IN SURVEY

PARTICIPANT CATEGORY	DEFINITION OF CATEGORY
Project managers	project managers of a project which include software line managers, each of whom had a first hand knowledge of the major problems of SEPM.
Project individuals	individual programmers/analysts who worked on projects.
Technical leaders	people who are in a high position of influence in their company's data processing function, highly visible authors or speakers on computer science, particularly data processing management; and authors of texts, papers, or reports on project management.
R&D personnel	people who through their business or avocation were interested in furthering the state-of-the-art in project management, or in perhaps a few cases, some other aspect of computer science.
Educators	usually university professors; however, some non-university educators were included.

5

TABLE IV  
SUMMARY OF PART A RESULTS—IMPORTANCE OF PROBLEM

INDEX NR	MAJOR ISSUE (Short-Title)	RESULTS IN PERCENT PROBLEM BY PARTICIPANT CATEGORY					
		COMPOSITE	PROJ MGRS	PROJ INDS	TECH LDERS	R&D	EDUCATORS
	NUMBER REPORTED	294	72	62	82	65	77
1	plan requirements	97	100	97	96	91	93
2	plan success	82	75	86	83	83	80
3	plan project	90	92	85	89	91	92
4	plan cost	88	92	86	84	83	85
5	plan schedule	94	97	93	91	92	93
6	plan design	72	71	67*	66*	72	75
7	plan test	79	85	77	73	69*	86
8	plan maintainability	57*	63*	61*	60*	70	72
9	plan warranty	74	72	70	72	70	78
10	plan control	61*	54*	60*	54*	63*	69*
11	organize type	46*	51*	40*	42*	41*	48*
12	organize accountability	81	75	72	86	69	88
13	staff project manager	77	73	82	72	70	77
14	direct techniques	59*	58*	50*	48*	57*	66*
15	control visibility	58*	65*	62*	71	77	74
16	control reliability	85	90	84	78	79	86
17	control maintainability	76	76	67*	71	82	82
18	control goodness	62*	60*	62*	60*	55*	65*
19	control programmers	78	77	70	78	81	84
20	control tracing	67*	67*	55*	70*	69*	68*

\* Results were inconclusive

survey results was to determine how different groups of people answered the individual questions. To do so, the participants were divided into the five categories shown in Table III. These five groups were selected because of their potential for conflicting views. Do project managers view the major issues of SEPM differently than project individuals? Do R&D personnel hold different beliefs than project managers? Do universities recognize the major problems so they can direct their teaching and research to those areas? And finally, what do the technical leaders believe?

The results of the survey are summarized in Tables IV-VI along with the applications of the 30/40/30 rule. Each major survey group is scored separately so that differences of opinion between groups can be observed. Parts "A," "B," and "C" are all shown separately. Part "D," in which respondents offered

solutions to the problems in English prose, is presented in the next section.

For part "A" of the survey, each respondent was asked to judge the relative importance of the hypothesized problem. For all groups at least 30 percent believed each proposition to be an important problem, so that none of the 20 issues can be discounted. For the six categories in Table IV there are a total of 120 figures shown (6 columns X 20 figures per column). Of these 120, only six fall below 50 percent, and only 15 fall below 60 percent. When all respondents are combined (COMPOSITE column), 13 of the 20 issues were classified as definite problems, while seven were inconclusive. These numbers varied somewhat across the five subgroups. Project managers also classified 13 issues as definite problems. For project individuals the number was 11; for technical leaders it was 14, for research

TABLE V  
SUMMARY OF PART B RESULTS—NATURE OF PROBLEM

6

INDEX NR	MAJOR ISSUE (Short Title)	RESULTS IN PERCENT MANAGEMENT PROBLEM BY PARTICIPANT CATEGORY					
		COMPOSITE	PROJ MGRS	PROJ INDS	TECH LDERS	R&D	EDUCATORS
1	plan requirements	64	69	68	56	56	61
2	plan success	70*	74*	71*	69	67	59
3	plan project	87*	89*	83*	90*	86*	80*
4	plan cost	71*	76*	64	75*	73*	70*
5	plan schedule	68	71*	69	68	69	68
6	plan design	45	50	52	40	34	32
7	plan test	41	49	48	31	32	32
8	plan maintainability	44	47	46	45	44	52
9	plan warranty	52	47	47	43	50	54
10	plan control	79*	86*	85*	80*	80*	66
11	organize type	92*	94*	95*	88*	92*	79*
12	organize accountability	94*	97*	97*	93*	94*	89*
13	staff project manager	95*	98*	96*	94*	94*	90*
14	direct techniques	89*	86*	89*	88*	90*	88*
15	control visibility	74*	78*	80*	73*	66	57
16	control reliability	23†	22†	27†	22†	16†	20†
17	control maintainability	36	42	43	32	30	38
18	control goodness	27†	30	23†	25†	25†	33
19	control programmers	65	62	75*	64	66	60
20	control tracing	62	67	75*	60	59	50

\* Management problem.

† Technical problem.

TABLE VI  
SUMMARY OF PART C RESULTS—NATURE OF SOLUTION

INDEX NR	MAJOR ISSUE (Short Title)	RESULTS IN PERCENT MANAGEMENT SOLUTIONS BY PARTICIPANT CATEGORY					
		COMPOSITE	PROJ MGRS	PROJ INDS	TECH LDERS	R&D	EDUCATORS
1	plan requirements	61	64	67	53	59	55
2	plan success	66	70*	65	66	62	59
3	plan project	84*	86*	87*	87*	81*	80*
4	plan cost	66	67	66	65	65	65
5	plan schedule	64	70*	63	63	62	65
6	plan design	45	50	47	40	35	25
7	plan test	40	48	42	36	33	33
8	plan maintainability	45	48	51	47	44	50
9	plan warranty	50	53	51	43	47	51
10	plan control	76*	81*	81*	78*	82*	68
11	organize type	91*	93*	95*	88*	91*	85*
12	organize accountability	92*	95*	95*	90*	91*	89*
13	staff project manager	96*	97*	96*	94*	96*	91*
14	direct techniques	83*	80*	80*	85*	86*	89*
15	control visibility	72*	76*	77*	70*	60	59
16	control reliability	24†	22†	33	23†	22†	20†
17	control maintainability	36	42	46	33	23†	39
18	control goodness	22†	26†	26†	21†	20†	22†
19	control programmers	61	61	66	59	61	59
20	control tracing	59	65	67	56	53	53

\* Management solution.

† Technical solution.

and development personnel it was 14, and finally educators felt that 15 of the issues were actual problems.

Project individuals overall categorized only 11 of the issues as definite problems versus 13 for the total set of survey participants. In no case did the programmers feel that an issue was definitely an important problem when the overall populace was unsure. Project individuals also inconclusively judged Questions 6 (plan design) and 17 (control maintainability) while the overall populace labeled these as definite problems. We find it very intriguing that programmers who must live with the consequences of poor designs and must maintain badly developed software did not conclusively feel that these two issues were important problems. We cannot, of course, with the data obtained, account for these feelings.

Educators, more than any other group, believed these issues

to be important problems. A full 15 of these issues were so categorized by them. In no case were the educators unsure of the importance of an issue for which the overall populace was decided. Educators believed that Issues 8 (plan maintainability) and 15 (control visibility) were important problems even though the general populace was inconclusive about them. In fact, the educators and R&D personnel were the only groups who did believe that planning for maintenance is an important problem. Perhaps this indicates why commercial software is so hard to maintain. Industrial personnel at all levels are not convinced that planning for maintenance is an important problem. It is interesting to note that instead of portraying an ivory tower attitude of indifference to "real world" problems, academicians seem even more concerned with these issues than do the commercial software personnel who must live with them daily!

Perhaps the most interesting group is the project managers, those who presumably carry out SEPM functions on a daily basis. Their perspectives agree with the overall populace in every case.

Overall, planning activities seem to be almost universally seen as being the most important problems. The first ten problems all deal with planning activities. For seven of these, all groups unanimously agreed that they are important problems. Eight out of ten planning problems are accepted as important by the overall populace.

In part "B" respondents indicated the nature of the problem, managerial, technical or both, while in part "C" they indicated whether the solution would be managerial, technical, or both. Table V shows in detail how each group responded to the questions of part "B," while Table VI shows the corresponding detail for part "C."

The overall populace thought that only nine of these problems were definitely management problems, thought that nine mixed both management and technology, and classified the other two as technical problems. For project managers this changed to 10, 9, and 1, respectively, while for the other groups the corresponding figures are: project individuals—10, 8, 2; technical leaders—8, 10, 2; research and development—7, 11, 2; and educators—6, 13, 1. Hence, it seems as while we are correct in stating that these 20 hypothesized problems are indeed problems, the general populace and specific subgroups do not all agree that these problems are managerial. In fact, there was unanimous agreement among all groups that controlling reliability is a technological not a managerial problem, and nearly unanimous agreement about controlling the "goodness" of code. Hence, it seems that we have validated nine problems as being primarily managerial, nine problems as having mixed elements of both management and technology, and two problems as being technological rather than managerial.

Educators and R&D personnel, who develop technical solutions to problems as part of their normal job activities, seemed more likely to see at least some technical aspects in these problems than the other groups. Project managers, on the other hand, who deal with these problems from a managerial perspective, seemed more likely to view them as being managerial in nature. However, for no group was the shift in opinion very dramatic. Interestingly enough, project individuals were the only group who felt that controlling programmers is a managerial problem.

When we look at part "C" in which people expressed their beliefs in whether these problems can be solved through improvements in management or technology, there is an analogous trend towards improvement in management. For the overall populace, seven problems were felt to be solvable by improvements in management, while 11 were mixed between management and technology. It was felt that two problems could be solved by improvements in technology alone. Project managers had somewhat stronger beliefs that improvements in management would solve 9 of the 20 problems. This number drops to seven for project individuals and technical leaders, to six for research and development personnel, and to just five for educators. The consensus of opinion across groups is strongest here. For 14 of the 20 problems, all groups scored the same.

7  
TABLE VII  
COMMON SOLUTIONS TO MAJOR PROBLEMS OF SOFTWARE ENGINEERING PROJECT MANAGEMENT

INDEX	SOLUTIONS
A	Use or enforce (existing) . . . standards, procedures, and documentation.
B	Define and conduct R&D in . . . . .
C	Use (existing) software engineering techniques and tools.
D	Improve or insure requirement specifications to include . . . . .
E	Educate or train project managers in . . . . .
F	Use competent, experienced software development managers.
G	Analyze . . . . data from prior software developments to determine best method of . . . . .
H	Use (existing) software engineering project management methods and tools.
I	Plan and manage or develop . . . . as part of project management plan . . . . .
J	Educate, inform, or involve (top) management in . . . . .
K	Review or audit . . . . .
L	Use extensive test techniques and tools.
M	Select and define the correct (or best) organizational structure (or test team, or development team, or . . . ) for project environment and techniques.
N	Involve user or increase communication between user and developer in . . . . .
O	Divide project or task into programs, modules, or sub-tasks and . . . . .
P	Educate or train software developers in . . . . .
Q	Use a project control system to . . . . .
R	Use good descriptive documentation techniques.
S	Use configuration management or change control procedures.
T	Define or consider quality in terms of deliverables, i.e., reliability, maintainability, etc.
U	Use competent, experienced software developers.
V	Allow sufficient time for . . . . .
W	Use existing, commercial software system to . . . . .
X	Use an (automatic) reporting, or tracking system.
Y	Use any method, the results are not sensitive to . . . . .
Z	Other.

Only for Question 15 (control visibility) was there disagreement by more than one group.

V. PROPOSED SOLUTIONS

Part "D" of the questionnaire asked respondents how they would or did solve the stated problem. Almost 4500 individual answers were given by the 294 people who returned the survey. Because these answers are in free-format English, it is impossible (and pointless) to list all of them separately. Instead we created a number of categories and grouped answers into those categories as shown in Tables VII and IX. The categories were constructed from a careful examination of the answers, and were not predefined or created independently of the survey.

The survey question form left only a small amount of room for a response to part "D." Hence, all of the responses are, unfortunately, quite vague. However, we feel that they still provide insight into what many people believe are solutions for the 20 major issues of SEPM. Table VIII shows some statistics on the more common solutions proposed. The questions are grouped into the five management activities of planning, organization, staffing, directing, and controlling, as was done in Table I. The figure for a particular solution category and man-

8

TABLE VIII  
STATISTICS ON COMMON SOLUTIONS

SOLUTION INDEX	PLAN	PCT OF TOTAL SOLUTIONS PER MGT FUNCTION				TOTAL
		ORGN	STAFF	DIR	CONTR	
A	11.3	16.2	7.0	28.2	22.7	15.4
B	10.8	2.7	5.5	10.7	16.0	11.1
C	6.6	--	--	--	9.0	6.0
D	8.6	--	--	--	3.4	5.5
E	4.3	5.0	17.2	14.6	2.5	5.0
F	3.9	6.9	16.4	9.7	2.6	4.8
G	3.9	--	--	10.7	5.8	4.1
H	3.9	--	--	--	6.4	3.8
I	4.5	1.9	--	--	0.9	2.8
J	3.6	--	13.3	--	0.1	2.6
K	1.5	--	3.1	--	5.2	2.4
L	3.3	--	--	--	0.6	1.9
M	0.2	16.2	--	--	0.3	1.9
N	2.8	0.8	--	--	0.1	1.6
O	1.6	--	--	--	2.2	1.4
P	2.0	--	--	--	1.0	1.4
Q	1.4	2.3	--	--	0.1	1.0
R	0.5	--	--	--	1.7	0.7
S	1.1	--	--	--	--	0.6
T	0.8	--	--	--	0.4	0.6
U	1.0	--	--	--	--	0.5
V	0.7	--	--	--	0.4	0.5
W	0.8	--	--	--	--	0.4
X	0.3	--	--	--	0.7	0.4
Y	0.1	0.8	--	2.9	0.1	0.3
Z	20.3	47.3	37.5	23.3	17.3	23.3

agement activity indicates what percentage of the total solutions presented for that activity fell into that solution category. For example, 11 percent of the solutions for "planning" activities could all be classified as category A, i.e., "use or enforce (existing) . . . standards, procedures, and documentation." Many of the more interesting solutions which did not cross over several issues are shown in Table IX.

Most of the categories listed account for only a small percentage of the total answers. However, in a few cases, a sizeable percentage of the total number of answers falls into one category. For example, 28 percent of the solutions related to directing questions essentially said to use or enforce standards, procedures, and documentation. Twenty-three percent of the solutions for control problems fell into this same category. Fifteen percent of the solutions for problems of directing fell into category H, i.e., educating and training project managers. Interestingly, category B, (define and conduct R&D in an area) never exceeded 16 percent.

The general conclusion to be drawn from the multitude of answers is that there is no consensus today as to how to solve these problems, although a sizeable percentage of the respondents occasionally supported a single solution for a single managerial activity. The fact that there is no consensus may in part be due to the unstructured format in which respondents stated their solutions. Perhaps if we had specifically listed a number of options, a stronger consensus might have been possible. The survey does show that, at least without a list of candidate solutions specifically presented, software engineering personnel do not generally agree on how to solve these major problems.

TABLE IX  
UNIQUE SOLUTIONS TO MAJOR PROBLEMS OF SOFTWARE ENGINEERING PROJECT MANAGEMENT

PROBLEM	WEIGHT	SOLUTION TO PROBLEM
Plan Requirements	9	Use iteration of the requirements with solutions.
Plan Success	3	Require management to define objectives in terms of quality desired
	2	Establish success priority criteria
Plan Cost	5	Develop a truthful, accurate cost accepted by management/customers and manage to it
	2	Allow for contingencies
Plan Schedule	5	Develop a truthful, accurate schedule accepted by management/customers and manage to it
	5	Allow for contingencies
Plan Design	1	Perform alternative analysis to select best techniques and tools
Plan Test	4	Design better software (to be tested)
Plan Warranty	4	Use follow-on maintenance support (include agreeing on price before developing software)
	1	Use follow-on maintenance contract
	1	Make it a legal problem not a software problem
Organize Type	11	Use organizational structure directed by top management (or staff)
	7	Use a project organization
	6	Use a matrix organization
	3	Use existing organizational structure
	3	Let the project manager select the organization type
	1	Give project manager full authority for project
Organize Account	19	Make specific assignment of work to software developers (using WBS)
	11	Define clearly each work package
	1	Give project manager full authority for project
Staff Project Manager	13	Have a selection process that selects good and eliminates poor project managers
	9	Use a selection criteria that is based on management abilities, not technical abilities (or availability)
	4	Establish job performance standards
	2	Promote software developers (who have demonstrated competence as managers)
	2	Have available a pool of project managers to select from
Direct Techniques	8	Use best judgment (individual problem)
	1	Promote successful software engineers to project managers
Control Reliability	2	Just get the programs to work (reliability not important)
Control Goodness	4	Just get the programs to work, that is sufficient
Control Programmers	5	Develop and use job performance standards for software developers
	2	Measure the results of the software development
	2	Keep records on productivity (eliminating low producers)

## VI. UNIVERSITY PROGRAMS ON SEPM

To augment the data collected in the original survey, modified questionnaires were mailed to over 100 universities including all institutions which grant Ph.D. degrees in computer science, and all computer science departments in the U.S. These questionnaires sought the opinions of professors on the same twenty issues, but more importantly attempted to determine how much emphasis is placed on these issues in classroom instruction.

The response rate was a somewhat disappointing 20 percent.

Follow-up phone calls to some of the schools which did not respond revealed that in some cases schools did not return the questionnaire because they do not offer any courses in which SEPM is covered. A further examination of course catalogs for over two dozen universities which did not respond showed that in only a handful of cases do these universities offer any courses which could be identified from the catalog descriptions as pertaining to the principles of SEPM. Although this survey is incomplete, it appears from the data gathered that few schools offer courses on software engineering principles beyond those of "structured programming." Less than 10 percent of the schools offer any courses which present a substantial amount of material on SEPM.

The questionnaires were "course" oriented. Each department was asked to identify its software engineering courses. No definition of "software engineering" was provided. It turned out that there was wide variation in how that term was interpreted. Some institutions interpreted virtually every software related course as being a software engineering course, beginning with introductory programming. Other departments only classified those advanced courses dealing with "software reliability," "software testing," and other more specialized topics as falling under software engineering.

For each software engineering course, the course content was identified, allowing us to determine whether any time was spent discussing SEPM. Those classes in which SEPM was not addressed are ignored in the data shown here. Only courses which have identifiable components dealing with SEPM are reported. Hence, courses dealing only in "structured programming practices" are not counted.

For each software engineering course in a department's curriculum, a separate questionnaire was completed which contained information pertinent to that course. The 20 problems were identical to those sent to the original group. However, instead of being asked to answer four parts for each problem, the professor answered only two. Part "A" was the same as before. This allowed us to gauge the relative importance which the professor attributed to each issue. Part "B" specifically asked to what extent this problem was covered in the class. The four options were: "a great deal," "somewhat," "very little," and "not at all." Almost without exception, professors seemed to feel that the problem is more important than the coverage actually allocated to it in the classroom. If we accept the idea that critical problems should be covered a great deal in class, important problems covered somewhat, problems which are not important covered very little, and issues which are not problems should not be covered at all, then a natural correspondence between the opinions of part "A" and the coverage indicated in part "B" emerges. This correspondence is shown in Table X. A negative number in the column labeled "A-B" indicates that an issue is inadequately covered by the professor's own admission. The more negative the number, the more inadequate the coverage. To compute the numbers shown, the answers of part "A" were numbered from 1 (for critical) to 4 (for not a problem), while those of Part "B" were numbered similarly (1 = a great deal, 4 = not at all). If a professor felt the question was improperly stated, his response is not counted in the table.

We phoned a number of the respondents to find out why such

9

TABLE X  
IMPORTANCE VERSUS COVERAGE

INDEX No.	MAJOR ISSUE (short title)	MEAN "A"	MEAN "B"	A - B
1	plan requirements	1.4	1.8	-0.4
2	plan success	1.9	2.9	-1.0
3	plan project	1.6	2.1	-0.5
4	plan cost	1.8	2.4	-0.6
5	plan schedule	1.9	2.4	-0.5
6	plan design	2.2	3.1	-0.9
7	plan test	1.9	2.1	-0.2
8	plan maintainability	1.8	1.8	0.0
9	plan warranty	1.6	2.8	-1.2
10	plan control	2.2	3.1	-0.9
11	organization type	2.6	3.4	-0.8
12	organization accountability	2.1	2.9	-0.8
13	staff project manager	2.1	3.2	-1.1
14	direct techniques	2.2	3.1	-0.9
15	control visibility	2.5	2.8	-0.3
16	control reliability	1.6	2.6	-1.0
17	control maintainability	1.9	2.2	-0.3
18	control goodness	2.2	2.1	+0.1
19	control programmers	2.1	3.0	-0.9
20	control tracing	2.5	3.0	-0.5

a disparity exists between the believed importance of an issue and the amount of coverage it receives in the classroom. While this sampling is small, we feel it provides some insight into the major issues of SEPM. There were three primary reasons cited for not covering these issues more:

- 1) lack of expertise,
- 2) lack of texts and other teaching materials, and
- 3) inappropriate for computer science departments.

Some of the professors stated that they had little or no managerial experience. As such, they felt that they lacked the expertise required to teach about SEPM. Instead, they concentrated on the more technical issues which they were more comfortable with and in which they had industrial experience. Others cited a lack of suitable textbooks in this area, arguing that for undergraduate classes it is very difficult to teach from a collection of notes or papers, and that it is possibly more effort on their part than it is worth to create such a collection. The reason they felt there were no textbooks which adequately handled the material was because the area is so new and because these issues indeed are problems so there are no solutions to write about! Perhaps the most interesting reason cited for not teaching more SEPM material is that some professors felt the material was not appropriate for a computer science department. Because the material is managerial, these professors believed that it should be taught in business schools within the university. When asked whether, in fact, this material was currently being taught in the business school on their campus, the answer was invariably no.

## VII. SUMMARY AND CONCLUSIONS

Software engineering project management has been the focus of much recent attention because of the enormous penalties incurred during software development and maintenance resulting from poor management. To date there had been no comprehensive study performed to determine the most significant issues of SEPM, their relative importance, or the research directions necessary to solve them. We conducted a major survey of

individuals from all areas of the computer field to determine the general consensus on SEPM problems.

Twenty hypothesized problems were submitted to several hundred individuals for their opinions. We believe that the experimental evidence supports the conclusion that at least 13 of the 20 hypothesized problems actually are important problems which face software engineering project managers. None of the other seven issues has been discounted, but opinion is more mixed on their relative importance.

A number of potential solutions or research directions were indicated by the respondents which, if followed, the respondents believe would lead to solutions for these problems. The task facing industry, government, and the universities is to solve these problems and to disseminate these solutions for the widespread benefit of the computing field.

#### ACKNOWLEDGMENT

Programmer and analyst support was provided by: B. J. Nieland, L. M. Hanger, R. D. Heckler, G. Collins, J. W. Robino, L. A. Morris, D. E. Sturdevant, and L. H. Jones.

Secretarial, proofreading, and composing support was provided by numerous people: B. E. McPheeters, T. Hamilton-Ricketts, M. L. Mueggenburg, W. Antwiler, and H. Antwiler.

Editing, proofreading, and overall grammatical corrections were provided by M. Smith and S. R. Green.

#### REFERENCES

- [1] J. D. Aron, "Estimating resources for large programming systems," in *Proc. NATO Conf. Software Eng. Concepts and Techniques*, P. Naur, B. Randell, and J. N. Buxton, Eds. New York: Petrocelli/Charter, 1976, pp. 206-217.
- [2] R. H. Austing, B. H. Barnes, D. T. Bonnette, G. L. Engel, and G. Stokes, Eds., "Curriculum '78," *Commun. Ass. Comput. Mach.*, vol. 22, Mar. 1979.
- [3] Statement of work, "Reliable software," BMDATC, Huntsville, AL, Rep. SW-A-44-74, Sept. 25, 1974.
- [4] —, "Data processing system requirements," BMDATC, Huntsville, AL, Rep. SW-A-88-75, Dec. 9, 1974.
- [5] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1227-1230, Dec. 1976.
- [6] —, "Software and its impact: A quantitative assessment," The Rand Corp., Santa Monica, CA, Rep. TR-P-4947, Dec. 1972.
- [7] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. 2nd Int. Conf. Software Eng.*, IEEE Comput. Soc., Oct. 1976, pp. 592-605.
- [8] F. P. Brooks, Jr., "The mythical man-month," *Datamation*, vol. 20, Dec. 1974.
- [9] J. D. Cooper, "Corporate level software management," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 319-326, July 1978.
- [10] B. C. DeRoze, "DOD defense system software management program" (Letter), Office of the Assistant Secretary of Defense, Washington, DC, Mar. 1, 1976.
- [11] B. C. DeRoze and T. H. Nyman, "The software life cycle—A management and technological challenge in the Department of Defense," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 309-318, July 1978.
- [12] *Findings and Recommendations of the Joint Logistics Commanders Software Reliability Work Group*, vol. 1, Executive Summary, Nov. 1975.
- [13] Invited workshop "Organizing ADP projects," cosponsored by: Nat. Bureau of Standards, IEEE Comput. Soc., and Federal Interagency Committee on Automat. Data Processing, June 13-14, 1978.
- [14] P. F. W. Keen and E. M. Gerson, "A politics of software system design," *Datamation*, vol. 23, pp. 80-84, Nov. 1977.
- [15] S. P. Keider, "Why projects failed," *Datamation*, vol. 20, pp. 53-55, Dec. 1974.
- [16] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.
- [17] —, *Software Tools*. Reading, MA: Addison-Wesley, 1976.
- [18] P. J. Klass, "NORAD data system has 100% overrun," *Aviation Week Space Technol.*, vol. 109, pp. 61-63, Oct. 30, 1978.
- [19] K. Kolence, "Software engineering management and methods" (Discussion), in *Proc. NATO Conf., Software Eng. Concepts and Techniques*, P. Naur, B. Randell, and J. N. Buxton, Eds. New York: Petrocelli/Charter, 1976, p. 13.
- [20] M. Lipow and T. A. Thayer, "Prediction of software failures," in *Proc. 1977 Annu. Reliability and Maintainability Symp.*, 1977, pp. 1-6.
- [21] J. H. Manley, "Embedded computer system software reliability," *Defense Management J.*, vol. 11, p. 13-18, Oct. 1975.
- [22] R. McCarthy, "Applying the technique of configuration management to software," *Defense Management J.*, vol. 11, pp. 23-28, Oct. 1975.
- [23] R. E. Merwin, "Software management: We must find a way" (Guest Editorial), *IEEE Trans. Software Eng.*, vol. SE-4, pp. 307-308, July 1978.
- [24] B. Miller, "Avionics problems bar debate of F15 with TAC," *Aviation Week Space Technol.*, vol. 103, pp. 23-25, Dec. 1975.
- [25] P. Naur, B. Randell, and J. N. Buxton, Eds., *Software Engineering: Concepts and Techniques*. New York: Petrocelli/Charter.
- [26] J. L. Ogden, "Designing reliable software," *Datamation*, vol. 18, pp. 71-78, July 1972.
- [27] A. J. Perlis, "Software engineering education" (Discussion), in *Proc. NATO Conf., Software Eng. Concepts and Techniques*, P. Naur, B. Randell, and J. N. Buxton, Eds. New York: Petrocelli/Charter, 1976, pp. 199-206.
- [28] Rome Air Development Center, unpublished R&D program in software cost reduction (FY), 1977.
- [29] S. R. Ruth, "What can the Navy learn from ALS?," unpublished document, 1974.
- [30] *Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's* (CCIP-85). Highlights, vol. 1, SAMSO Tech. Rep. 72-141, Apr. 1972.
- [31] J. I. Schwartz, "Analyzing large-scale system development," in *Proc. NATO Conf., Software Eng. Concepts and Techniques*, P. Naur, B. Randell, and J. N. Buxton, Eds. New York: Petrocelli/Charter, 1976, pp. 260-275.
- [32] J. B. Slaughter, "Understanding the software problem," in *Proc. Symp. High Cost of Software*, J. Goldberg, Ed. Menlo Park, CA: Stanford Res. Inst., Sept. 17-19, 1973, pp. 41-52.
- [33] J. M. Spier, "Software malpractice—A distasteful experience," *Software—Practice and Experience*, vol. 6, pp. 293-299, 1976.
- [34] R. H. Thayer, "The Rome Air Development Center R&D program in computer languages and software engineering," Rep. RADC TR 74-80, Apr. 1974.
- [35] D. A. Walsh, "Structured testing," *Datamation*, vol. 23, pp. 111-118, July 1977.

Richard H. Thayer (S'61-M'63-SM'72-S'76-M'77-S'78-M'78), photograph and biography not available at the time of publication.

Arthur B. Pyster (M'76), photograph and biography not available at the time of publication.

Roger C. Wood (M'68), photograph and biography not available at the time of publication.

# A Software Maintainability Evaluation Methodology

DAVID E. PEERCY 11

**Abstract**—This paper describes a conceptual framework of software maintainability and an implemented procedure for evaluating a program's documentation and source code for maintainability characteristics. The evaluation procedure includes use of closed-form questionnaires completed by a group of evaluators. Statistical analysis techniques for validating the evaluation procedure are described. Some preliminary results from the use of this methodology by the Air Force Test and Evaluation Center are presented. Areas of future research are discussed.

**Index Terms**—Evaluation by questionnaire, evaluation reliability, quality metrics, software engineering, software maintainability evaluation, software quality assurance.

## I. INTRODUCTION

THE Air Force Test and Evaluation Center (AFTEC) has been developing a methodology for evaluating the quality of delivered software systems as part of its directed activity of operational test and evaluation (OT&E). Thayer [3] has reported the initial approach for a software maintainability evaluation methodology. The BDM Corporation has completed a technical directive for AFTEC to review this methodology, analyze the results of 18 different program evaluations which used the methodology, and recommend appropriate changes to the methodology. This paper summarizes the revised methodology from this effort.

Because of the number of software systems to be evaluated, the variability (language, computer, functions) of the software to be evaluated, and the limited state of the art in practical automated evaluation tools, AFTEC's software evaluation procedure has been based on the completion of closed-form questionnaires. The methodology defines a conceptual framework for the software characteristics from the user-oriented level to the software product level and an evaluation procedure whereby the identified product characteristics can be measured. The measures, or software metrics, are then normalized through evaluation-specific weights to provide the necessary evaluation maintainability measures.

The primary objective of the software maintainability evaluation is to collect enough specific information to identify for which parts of the software and for what reasons maintainability may be a problem. A secondary objective is to assess the effectiveness of the evaluation process itself. A future goal is to validate maintainability scores against an actual field maintenance level of effort.

Manuscript received February 4, 1980; revised January 21, 1981. This work was supported by AFTEC Technical Directive 120 of Contract F29601-77-C-0082.

The author is with the BDM Corporation, Albuquerque, NM 87106.

The major evaluation assumptions are as follows:

- maintainability considerations remain essentially the same from program to program,
- evaluators must be knowledgeable in software procedures, techniques, and maintenance, but need not have a detailed knowledge of the functional area of the program,
- a minimum of five independent evaluators will be used to provide acceptable confidence that the metrics (evaluator average scores) are a sound measuring tool,
- the random sampling of the program modules for evaluation provides conclusions which hold for the general population of all program modules.

The main features of the software maintainability evaluation are the following.

- The maintainability model is primarily based on the models in Thayer [3], Boehm [1], and Walters [4]. The evaluation process consists of a set of evaluators completing closed-form questionnaires on maintainability characteristics of program documentation and program source listings followed by automated processing of the evaluator responses and a careful manual analysis of all detected program and evaluation anomalies.
- The evaluation can be used at appropriate phases in the software development life cycle in addition to the operational maintenance phase.
- The evaluation is independent of any particular source language.
- The maintainability characteristics can be used as a quality assurance checklist.

The major results of this research effort have been to:

- provide a definitive evaluation methodology which is practical and immediately useful to AFTEC,
- reduce subjectivity and increase reliability of the evaluation,
- provide a conceptual framework which can be expanded both within maintainability and to other quality factors,
- provide a computer program for the automated processing and analysis of the evaluation data.

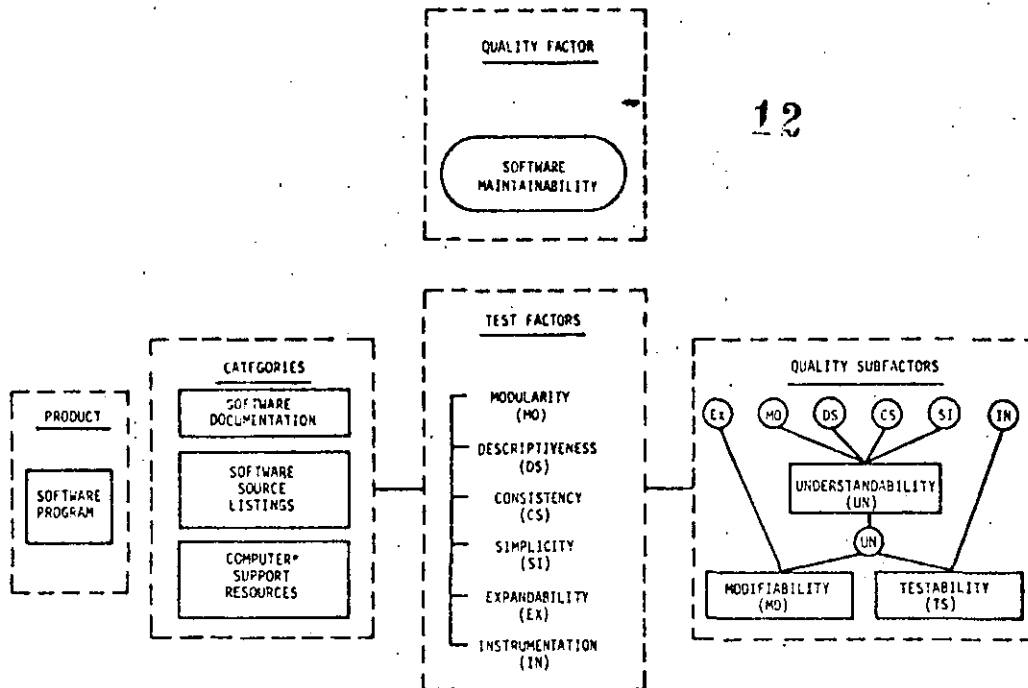
## II. CONCEPTUAL FRAMEWORK

The software maintainability evaluation methodology has the conceptual framework shown in Fig. 1. The associated definitions are in Table I.

### A. Quality Factors

The work of Boehm [1] and Walters [4] among others has established a set of user-oriented terms representing desired qualities of software. These terms, or quality factors, include maintainability, usability, correctness, human engineering,





12

Fig. 1. Elements of software maintainability.

TABLE I  
DEFINITIONS

<p><b>SOFTWARE:</b> Software consists of the programs and documentation which result from a software development process.</p> <p><b>SOFTWARE MAINTAINABILITY:</b> Software maintainability is a quality of software which reflects the effort required to perform the following actions:</p> <ol style="list-style-type: none"> <li>(1) Removal/correction of latent errors</li> <li>(2) Addition of new features/capabilities</li> <li>(3) Deletion of unused/undesirable features</li> <li>(4) Modification of software to be compatible with hardware changes</li> </ol> <p>Implicit in the above definition are the concepts that the software should be understandable, modifiable and testable in order to have effective maintainability.</p> <p><b>UNDERSTANDABILITY:</b> Software possesses the characteristics of <u>understandability</u> to the extent its purpose and organization are clear to the inspector.</p> <p><b>MODIFIABILITY:</b> Software possesses the characteristics of <u>modifiability</u> to the extent that it facilitates the incorporation of changes once the nature of the desired change has been identified.</p> <p><b>TESTABILITY:</b> Software possesses the characteristics of <u>testability</u> to the extent that it facilitates the establishment of verification criteria and supports evaluation of its performance.</p> <p><b>TEST FACTORS:</b> Software maintainability test factors are user-oriented general attributes of software which affect maintainability. The set of test factors includes: modularity, descriptiveness, consistency, simplicity, expandability, and instrumentation.</p> <p><b>MODULARITY:</b> Software possesses the characteristics of <u>modularity</u> to the extent that a logical partitioning of software into parts, components, and modules has occurred.</p> <p><b>DESCRIPTIVENESS:</b> Software possesses the characteristics of <u>descriptiveness</u> to the extent that it contains information regarding its</p>	<p>objectives, assumptions, inputs, processing, outputs, components, revision status, etc.</p> <p><b>CONSISTENCY:</b> Software possesses the characteristics of <u>consistency</u> to the extent the software products correlate and contain uniform notation, terminology and symbology.</p> <p><b>SIMPLICITY:</b> Software possesses the characteristics of <u>simplicity</u> to the extent that it lacks complexity in organization, language, and implementation techniques and reflects the use of singularity concepts and fundamental structures.</p> <p><b>EXPANDABILITY:</b> Software possess the characteristics of <u>expandability</u> to the extent that a physical change to information, computational functions, data storage or execution time can be easily accomplished.</p> <p><b>INSTRUMENTATION:</b> Software possesses the characteristics of <u>instrumentation</u> to the extent it contains aids which enhance testing.</p> <p><b>SOFTWARE DOCUMENTATION:</b> Software documentation is the set of requirements, design specifications, guidelines, operational procedures, test information, problem reports, etc. which in total form the written description of the program(s) output from a software development process.</p> <p><b>SOFTWARE SOURCE LISTINGS:</b> Software source listings are the implemented representation (listing) described through a source computer language of the program(s) output from a software development process.</p> <p><b>COMPUTER SUPPORT RESOURCES:</b> Computer support resources include all the resources (software, computer equipment, facilities, etc.) which support the software being evaluated.</p> <p><b>PROGRAM:</b> A program is a set of hierarchically related modules which can be separately compiled, linked, loaded and executed.</p> <p><b>MODULE:</b> A module is a set of "contiguous" computer language statements which has a name by which it can be separately invoked</p>
--	---

portability, reliability, and others. Although the quality factors may be the same syntactically among researchers, semantically they tend to have different interpretations. The definition in Table I of software maintainability reflects AFTEC's concern for acquiring software which is understandable (required locations for modifications can be easily established), modifiable (enhancements or corrections can be made), and testable (the software is properly instrumented for testing once modifications have been made).

**B. Software Product/Categories**

Each software program (product) is separately evaluated and consists of a set of components called modules. A module may, in general, be at any conceptual level of the program.

For each program there are three categories which are evaluated for characteristics which affect maintainability: software documentation, software source listings, and the computer support resources. Only program deliverables are considered in an evaluation.

1) *Software Documentation:* The primary documentation used in this evaluation consists of the documents containing program design specifications, program test plan information and procedures, and program maintenance information. These documents may have a variety of physical organizations depending upon the particular application, although software standards attempt to reduce the variability [22]-[28]. The documents are evaluated both for content and for general physical structure (format).

2) *Software Source Listings*: The source listings represent the program as implemented, in contrast to the documentation which represents the program design or implementation plan. Source listings are also a form of program documentation, but for this maintainability evaluation a distinction is made.

The source listing evaluation consists of a separate evaluation of each specified module's source listing and the consistency between the module's source listing and the related written module documentation. The separate module evaluations are accumulated into an overall evaluation of the software source listings.

3) *Computer Support Resources*: Attributes and procedures for the evaluation of computer support resources are being developed and will be detailed in a separate report.

### C. Software Maintainability Test Factors

The maintainability of software documentation and source listings is a function of six attributes or test factors: modularity, descriptiveness, consistency, simplicity, expandability, and instrumentation. These test factors are defined in Table I. Discussions of their application in the evaluation of the documentation and source listings are given in the following paragraphs.

1) *Modularity*: It has been observed that software has been easiest to understand and change when composed of "independent" parts (sections, modules). Documentation and source listings are evaluated in relation to the extent their logical links show only a few, simple links to other parts (low coupling [9]) and contain only a few easily recognizable subparts which are closely related (high strength [9]). Parnas [10], [11] has described these concepts in different, but relatively equivalent terms.

2) *Descriptiveness*: It is important that the documentation contain useful explanations of the software program design. The objectives, assumptions, inputs, and outputs are desirable in varying degrees of detail in both documentation and source listings. The intrinsic descriptiveness of the source language syntax and the judicious use of source commentary greatly aids efforts to understand the program operation.

3) *Consistency*: The use of some standards and conventions in documentation, flowchart construction, I/O processing, error processing, module interfacing, and naming of modules/variables are typical reflections of consistency. Consistency allows one to easily generalize understanding. For example, programs using consistent conventions might require that the format of modules be similar. Thus, by learning the format of one module (preface block, declaration format, error checks, etc.) the format of all modules is learned.

4) *Simplicity*: The aspects of software complexity (or lack of simplicity) that are emphasized in the evaluation relate primarily to the concepts of size and primitives. The use of high order language as opposed to assembly language tends to result in a program simpler to understand because there are fewer discriminations which have to be made. There are certain programming considerations such as dynamic allocation of resources, recursive/reentrant coding which can greatly complicate the data and control flow. Real-time programs, because of the requirement for timing constraints and efficiency, tend to have more control complexity. The sheer bulk

of counts (number of operators, operands, nested control structures, executable statements, statement labels, decision parameters) will determine to a great extent how simple or complex the source code is [15]-[18].

5) *Expandability*: Software may be reasonably understandable but not easily expandable. If the design of the program has not allowed for a flexible timing scheme or a reasonable storage margin, then even minor changes may be extremely difficult to implement. Parameterization of constants and basic data structure sizes usually improves expandability. It is also very important that the documentation include explanations of how to make increases/decreases in data structure sizes or changes to the timing scheme. The limitations of such program expandability should be clear. The numbering schemes for documentation narrative and graphic materials must be carefully considered so that physical modifications to the documentation can be easily accomplished when necessary.

6) *Instrumentation*: For the most part, the documentation is evaluated by how well the program has been designed to include test aids (instruments), while the source listings are evaluated by how well the code seems to be implemented to allow for testing through the use of such test aids. The software should be designed and implemented so that instrumentation is imbedded within the program, can be easily inserted into the program, is available through a support software system, or is available through a combination of these capabilities.

### D. Software Characteristics

Each test factor has a set of software-level characteristics which serve to define the test factor within the context of the software product category being evaluated. Characteristics were identified and grouped so as to minimize the overlap among the test factors and balance the number of characteristics across the test factors. Characteristics for the documentation and source listings were identified primarily from Thayer [3], Boehm [1], Walters [4], Kernighan and Plauger [7], Myers [9], Parnas [10], [11], Miller [19], Halstead [15], [16], McCabe [18], Yeh [12], and various documentation standards [22]-[28].

A fixed scale for all evaluation responses was chosen and closed-form questionnaires for documentation and source listings were designed based on the identified characteristics. In order to minimize subjectivity, increase the evaluation reliability, and provide for a more efficient evaluation process, a detailed evaluation guidelines handbook [33] was developed. The handbook contains background methodology, the evaluation questionnaires, and a set of guidelines for interpreting the terminology and potential responses for each question. A computer program was developed to aid the analysis of the evaluator responses.

## III. EVALUATION PROCEDURE

The software evaluation procedure involves four distinct phases as shown in Fig. 2: planning, calibration, assessment, and analysis.

During the planning phase, the AFTEC Software Test Manager (STM) and the selected Software Assessment Team (SAT) Chairman establish evaluator teams, each consisting of

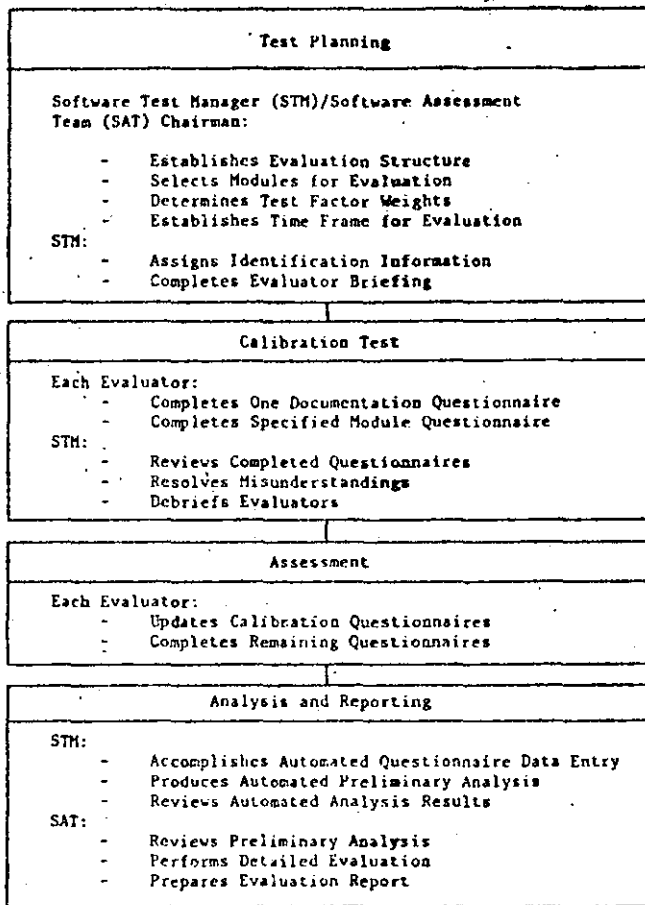


Fig. 2. Maintainability evaluation procedure.

at least five evaluators knowledgeable in software maintenance. The SAT chairman may or may not be one of the evaluators. The evaluators are preferably persons who will be responsible for maintaining some part of the software being evaluated. The program/module hierarchy is established and a set of representative modules is selected for each program to be evaluated. At least 10 percent of the modules in a program are randomly selected for evaluation. Specific test factor (attribute) weights are also determined at this time and the schedule for the evaluation is established. The software test manager briefs the evaluator teams on the procedures and assigns the necessary identification information for this specific evaluation.

The function of the calibration test is to ensure a reliable evaluation through a clear understanding of the questions and their specific response guidelines on each questionnaire. Each evaluator completes a documentation and module source listing questionnaire. The completed questionnaires are reviewed to detect areas of misunderstanding and the evaluation teams are debriefed on the problem areas.

In the assessment phase, the evaluation teams update their calibration test questionnaires based on the results of the calibration debriefing. The teams then complete the remainder of their assigned documentation and module source questionnaires. It is estimated that each evaluator will take 4-6 hours to complete the documentation questionnaire and 1-3 hours to complete each module questionnaire.

In the analysis phase, the software test manager accomplishes

14

TABLE II  
TEST FACTOR QUESTION DISTRIBUTION

	MO	DS	CS	S1	EX	IN	GENERAL	TOTAL
Documentation	12	24	9	12	9	10	7	83
Source Listings	14	21	14	16	9	8	7	89

TABLE III  
EXAMPLE QUESTIONS

<b>(Documentation)</b>	
<b>Format Modularity</b>	
Note: The following questions relate to how the documentation has been physically formatted into functional parts.	
1.	Program documentation includes a separate part for the description of program external interfaces.
2.	Program documentation includes a separate part for the description of each major program function.
3.	Program documentation includes a separate part for the description of the program global data base.
<b>Processing Modularity</b>	
Note: The following questions relate to how the program control and data flow has been designed for functional use.	
8.	The program control flow is organized in a top down hierarchical tree pattern.
9.	Program initialization processing is done by one (set of) module(s) designed exclusively for that purpose.
<b>(Source Listings)</b>	
<b>Size Simplicity</b>	
Note: The following questions relate to various "counts" which reflect the amount of information which must be assimilated to understand a module.	
62.	The number of expressions used to control branching in this module is manageable.
63.	The number of unique operators in this module is manageable.
64.	The number of unique operands in this module is manageable.
65.	The number of executable statements in this module is manageable.
<b>General Questions</b>	
83.	Modularity is reflected in this module's source listing contributions to the maintainability of this module.

the conversion and initial data processing of the questionnaire data. This preliminary analysis is then reviewed and corrected, if necessary. The statistical summaries are then returned to the SAT for detailed evaluation and preparation of the final report.

A. Example Questions

Each evaluator is supplied with a documentation questionnaire, source listing questionnaire, evaluation response forms, and an evaluator guidelines handbook. The number of questions for each of the questionnaires and each of the test factors is summarized in Table II. Some of the questions are illustrated in Table III. Note the "general" question 83. Each test factor has such an associated general question. In future analysis of the methodology, test factor characteristics (scores) will be regressed against the general-question (score) across all program modules and all programs. The guidelines for one of the sample questions are illustrated in Table IV.

B. Response Form

The form on which an evaluator records responses to questions is processed through an optical scanner. There are three "blocks" on this form: descriptive identification block, numerical identification block, and evaluator response

TABLE IV  
EXAMPLE OF QUESTION GUIDELINES

15

Question Number 5-62

**QUESTION:** The number of expressions used to control branching in this module is manageable.

**CHARACTERISTIC:** Simplicity (size simplicity).

**Explanation:** The count of control expressions is closely related to the number of independent cycles in a module. The more control expressions there are, the more complex the control logic tends to be.

**EXAMPLES:** The following examples indicate how to count the control expressions:

CONTROL STRUCTURE	STATEMENT	CONTROL EXPRESSION	COUNT
Decision	IF (A OR B) GO TO 10	A;B	2
	IF (A AND B) GO TO 10	A;B	2
	IF (C;GT.D) GO TO 10	C;GT.D	1
	IF (A AND B) OR (C;GT.D)	A;B;C;GT.D	3
	GO TO 10	I=1;I=2;I=3	2
	CASE (I) OF	(Alternatives) (number of alternatives less one)	
1: A			
2: B			
3: C			
END CASE			
Iteration	DO 10 I=1, 10	1;LT.1	
	A	I;GT.10	2
	10 CONTINUE		

**GLOSSARY:** Control expression: IF, CASE, or other decision control expression. DO, DO-WHILE, or other iterative control expression.

**SPECIAL RESPONSE INSTRUCTIONS:** The following guidelines will anchor A and F responses. But are fairly subjective (especially the F anchor). The guidelines for the A response is suggested from other independent research. Remember to count all repetitions of the same control expression also.

Answer A if count < 10.  
Answer F if count ≥ 50.

weights at the discretion of the AFTEC Test Manager and SAT Chairman, but raw scores will also be retained.

Assessment of the evaluation process itself is partially based on six measures: agreement, outliers, response distribution, standard deviation, regression, and question reliability.

Agreement on a question is calculated using the following formula:

$$AG = \frac{1}{NE} \sum_{i=0}^{NS} NR_i / 2^i$$

where  $AG$  is the agreement factor,  $NS$  is the number of unit steps in the scoring scale,  $NR_i$  is the number of responses that are  $i$  steps from the mode, and  $NE$  is the number of evaluators (responses).

If there is no mode, then the scale value closest to the mean and with at least as many responses as any other scale value is used as the "mode." As an example, with five responses of  $B, C, C, C, E$ , the mode is  $C$  and  $AG = \frac{1}{5} (3/2^0 + 1/2^1 + 1/2^2) = 0.75$ .

Outliers are determined in a somewhat subjective (but logical) manner since neither the agreement factor nor standard deviation provide acceptably consistent measures. An outlier is any extreme response with a distance ( $DE$ ) from the next closest response such that  $DE/DT > 0.5$ , where  $DT$  = maximum distance between any two responses.

Response distribution is studied across all evaluations on a question-by-question basis to determine the validity of the general assumption of a normal response distribution. Such analysis can also be used to determine an experimental question weight. On an individual evaluation basis, the combination of agreement, outlier, and standard deviation analysis is used to pinpoint particular questions which have an unacceptable response distribution.

Regression analysis is used across all evaluations to study the validity of the test factor question groupings and to study the regression of test factor characteristic responses against the associated general test factor question response. Itzfeldt [2] presented some interesting related results using regression and factor analysis.

Reliability is a measure of consistency from one set of measurements to another. Reliability can be defined through error: the more (less) error, the lower (higher) the reliability. Since we can measure total variance, if we can estimate the error variance of a measure, we can also estimate reliability.

The statistical method for identifying error variance is Analysis of Variance (ANOVA). ANOVA allows the analyst to isolate the sources of variance within total variance. In the evaluation of module questionnaires, for example, the sources of variance are differences between the evaluators due to their differing backgrounds and expectations, differences in the characteristics of the modules, and unattributable differences due to error. Two-way analysis of variance allows a determination of all three variance sources. Mean-squares for raters, modules, and error are determined as measures of variance. Reliability is then calculated as 1.00 minus the proportion of mean-square error to mean-square modules.

If the reliability coefficient  $R$  is squared ( $R^2$ ), it becomes a

block. The descriptive identification block contains information which identifies the particular questionnaire type, system, subsystem, program, module, evaluator, date, and time to complete. This block is only used for a visual identification check and is not processed by the optical scanner. The numerical identification block contains numeric codes for the same information contained in the descriptive identification block. The evaluator response block contains a set of 10 responses (A-J) for each question up to 90 questions.

### C. Response Scale

The following response scale is used to answer each question:

- completely agree,
- strongly agree,
- generally agree,
- generally disagree,
- strongly disagree,
- completely disagree.

One of these responses *must* be given for each question. In addition, one or more of the following standardized comment responses can be selected:

- I had difficulty answering this question,
- a written comment has been submitted.

The responses g and h are not used. The responses a-f (equivalent numeric metric is 6 to 1) indicate the extent to which the evaluator agrees/disagrees with the question statement.

### D. Analysis Techniques

The maintainability metrics are the average scores across evaluators, test factors, product categories, and programs. Test factors, product categories, and programs can be given

## Editor's Note

**T**HE following paper represents an experiment in publishing for the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. In July 1978, I received the original paper and wrote to the Editor of the TRANSACTIONS.

"I would like your reaction to the following idea: I think that this paper would make an excellent contribution to the TRANSACTIONS if we could do the following:

- 1) Send copies to a few selected "commentators"
- 2) Get the commentators to write short, but not trivial, remarks about the design approach
- 3) Get the authors to respond briefly to the remarks
- 4) Publish paper, as is; remarks; and response to remarks.

I have seen this done in other journals, and with excellent results. I think it is particularly appropriate when we are looking at the *design* of a *programming tool*. Rather than have the authors hide all the design short-sightedness they may have had, they are willing to reveal the thought processes, with all their flaws. This is a real opportunity to learn from their mistakes, as well as from their good judgments."

Raymond Yeh accepted the idea in concept, and I sent out seven requests for comments. After a considerable delay, I received four very thoughtful and extensive comments. In addition, I had a dozen pages of my own comments. I felt that this response indicated the magnitude of the interest in the subject, and the approach, but I was worried about the volume of the material.

I wrestled with the problem for a long time, through a change in IEEEETSE editorship, and through a wait for a response from the three original authors. Most of the delay, though, was mine, for I could not see how to edit this volume of material to a more moderate size. Finally I made my decision, and if you are reading this it means that the editorial board accepted it. I would keep the general comments largely unedited, but leave out the voluminous detailed comments on the design. I made this decision in keeping with a principle we have learned over a decade of design reviews—there is no sense criticizing the details until you have agreement on the general concepts.

Naturally, there are details that display the general concepts in a way that a vague statement would not illuminate, but after surveying this material many times, I believe that what Software Engineering needs most, right now, is some discussion of the broadest principles. Therefore, I have removed detailed comments from the commentators' notes. An enormous amount of valuable information was set aside in this way, but I take full responsibility for placing the priority on what remains.

If there should be an upswelling of popular demand, then perhaps I can convince the editorial board to complete this project by printing the detailed comments. But let us put first things first.

GERALD M. WEINBERG

# The Annotated Assistant: A Step Towards Human Engineering

ANDREW SINGER, MEMBER, IEEE, HENRY LEDGARD, AND JON F. HUERAS, MEMBER, IEEE

Alice had been looking over his shoulder with some curiosity. "What a funny watch!"<sup>5</sup> she remarked. "It tells the day of the month, and doesn't tell what o'clock it is!"

"Why should it?" muttered the Hatter. "Does *your* watch tell you what year it is?"

"Of course not," Alice replied very readily: "but that's because it stays the same year for such a long time together."

"Which is just the case with *mine*," said the Hatter.

Alice felt dreadfully puzzled. The Hatter's remark seemed to her to have no sort of meaning in it, and yet it was certainly English. "I don't quite understand you," she said, as politely as she could.

"The Dormouse is asleep again," said the Hatter, and he poured a little hot tea upon its nose.

5. An even funnier watch is the Outlandish Watch owned by the German professor in Chapter 23 of *Sylvie and Bruno*. Setting its hands back in time has the result of setting events themselves back to the time indicated by the hands; an interesting anticipation of H. G. Wells's *The Time Machine*. But that is not all. Pressing a "reversal peg" on the Outlandish Watch starts events moving *backward*; a kind of looking-glass reversal of time's linear dimension.

One is reminded also of an earlier piece by Carroll in which he proves that a stopped clock is more accurate than one that loses a minute a day. The first clock is exactly right twice every twenty-four hours, whereas the other clock is exactly right only once in two years. "You *might* go on to ask," Carroll adds, "How am I to know when eight o'clock *does* come? My clock will not tell me." Be patient: you know that when eight o'clock comes your clock is right; very good; then your rule is this: keep your eyes fixed on the clock and the *very moment it is right* it will be eight o'clock."

Reprinted from *The Annotated Alice* by Martin Gardner. Copyright © MCMLX Martin Gardner by permission of Clarkson N. Potter, Inc.

—*The Annotated Alice*

*The Hatter's watch nicely illustrates the effect of idiosyncrasy in system design. Really, a watch could provide any number of features, but most watches designed for people put a high priority on telling the correct time of day: Thus, the Hatter's watch is an excellent example of bad human engineering. By human engineering we mean "the selection among design alternatives so as to relate to people." Carroll's stopped watch is the ultimate in poor human engineering because the user must do all the work.*

## INTRODUCTION

OVER the years many authors (Cooke and Bunt [4], Cuadra [5], Holt and Stevenson [9], Kennedy [11], Mann [14], Palme [16], Parsons [17], Sterling [20], Vanden-

berg [24], and Weinberg [25]) have made the case for human engineering of computer systems. As Sterling [20] and Holt and Stevenson [9] have pointed out, human engineering is something that must be integrated into the design process, i.e., *it cannot be grafted on later*. We believe that very few systems have been designed with first priority given to human factors. The work described here reflects a conscious attempt to design a computer system in which human considerations had top priority in the design process.

As part of our general attempt to limit the influence of implementation considerations, at the start we chose to complete the design of every system feature before undertaking any implementation. Thus far we have adhered to that decision, and at present we are working on a detailed formal description of the entire system.

For a variety of reasons, we undertook the task of developing a standard interactive environment for PASCAL programmers, although the use of PASCAL is incidental to our design. We were interested in assisting both naive and sophisticated programmers.

The effort was motivated by several concerns:

- 1) the development of a moderately powerful system that makes users more productive with less effort;
- 2) the need for a system that stimulates rather than dampens the enthusiasm of potential users;

Manuscript received July 28, 1978. This work was supported by the National Science Foundation and the U.S. Army Research Office, Durham, NC.

A. Singer was with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. He is now with E & L Instruments, Derby, CT 06418.

H. Ledgard was with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003. He is now with Human Factors Limited, Leverett, MA 01054.

J. F. Hueras was with the Department of Information and Computer Science, University of California, Irvine, CA 92717. He is now with Processor Sciences, Inc., Burlington, MA 01803.

3) a desire to create a system without the trappings of conventional computer concepts, terminology, and jargon;

4) the need for documentation embodying a light and friendly approach to users.

These goals are easy to talk about, but difficult to realize. After nearly two years' work, we produced a design carefully documented in the form of a User's Guide. This document itself was the result of considerable effort. The final design represents a large number of rewrites, perhaps ten, aimed at making the system more accessible to the user. We consider this document to be an example of our overall concern with the human engineering of the system as a whole.

In this paper, we present the User's Guide (with only one of its three Appendices) in annotated form. The notes are intended to illuminate the human engineering design considerations and to explain the principles motivating our decisions.

It is important to bear in mind that the state of our knowledge about what constitutes a "good" decision from a human factors point of view is rather primitive. Often our only guide is intuition based on experience. Ultimately, we believe that the results of existing experimental investigations in psychology and new human engineering experiments in computing will provide a solid basis for designs. Nevertheless, even without much data we must give current systems the benefit of the best knowledge that we have. This is the case here.

In a sense, whenever we build a system today we conduct a human factors experiment. Unfortunately, we are rarely in a position to extract any valuable data from these experiments because we do not set them up as such. In the design of the Assistant, we developed a number of general principles to guide our decisions. As people use the system we expect to discover that some of these are wrong. But at the very least, we know we will be able to learn from our experience. Given the limits of hard data, we believe that work must proceed in this way if we are to advance the state of human engineering in computer systems.

??

## THE CURRENT STATUS OF THE ASSISTANT

The design described here was frozen in the Summer of 1976. Since then we have been engaged in a variety of tasks related to it. Considerable effort has gone into a full formal definition of the Assistant. At present, a complete definition of its interactive behavior is done, and a partial definition of its semantics has been written. This work is described by Singer [18]. A text editor, HOPE, based on the Assistant's editing requests has been written in PASCAL and has been available on the University of Massachusetts time-sharing system. The stand-alone automatic prettyprinting program (also in PASCAL) mentioned in the notes has been available from the PASCAL User's Group and currently has been distributed to more than 100 installations. Continuing efforts are aimed at completing the formal definition of the Assistant and eventually building a prototype. All of these efforts, notably the design of the Assistant and its human factors considerations, have required about ten person-years of effort, which we consider as research.

In reading this paper, it is important to bear in mind that what we are discussing is the *design*, not the implementation, of a system that we believe will be well within the state of the art. Until the complete formal description or an actual implementation has been completed, it remains to be seen whether this is the case.

All of this activity has not been without its effects. A variant of HOPE has been the basis of an experiment examining one of the hypotheses evolved during the Assistant's design (see Ledgard, Whiteside, Singer, and Seymour [28]), and a recent monograph (Ledgard, Singer, and Whiteside [13]) describes some of our emerging beliefs. A number of specific second thoughts concerning the design of the Assistant have also emerged. Whether we will incorporate these into a Revised User's Guide remains to be seen; but for the sake of completeness, we have included these thoughts in the following annotations.

# A User's Guide to the PASCAL Assistant

## INTRODUCTION

*"assistant . . . 1. one who assists or gives aid and support; a helper; . . ."*

*—Random House Dictionary of the English Language*

*"automaton . . . 1. a mechanism that is relatively self-operating; esp. ROBOT  
2. a machine or control mechanism designed to follow automatically a predetermined sequence of operations or respond to encoded instructions 3. a creature who acts in a mechanical fashion . . ."*

*—Webster's New Collegiate Dictionary*

The Assistant<sup>1</sup> This section is your introduction to a little robot we have created called "The PASCAL Assistant." This Assistant can help you create, manipulate, and execute PASCAL programs. Like any creature, natural or artificial, the Assistant has its own ideas about things. Unlike ourselves, however, the Assistant's ideas are fairly fixed, and its intelligence is limited. As with any assistant, your understanding of it will make for a smooth working relationship.<sup>2</sup>

1. An important part of the human engineering of a system is the physical display and organization of its documents. Throughout the User's Guide we attempt to keep a layout that is both visually appealing and yet can be used for quick reference. Because the User's Guide is short, there is little need for an index. Instead, keywords and key phrases are given in

**Terminals** The Assistant exists as a collection of computer programs that run on a time-sharing computer. Since you and the Assistant interact solely by means of a teletype or some other interactive terminal, we will at times describe the Assistant's behavior in terms of what the terminal does. Because you may be using any one of a variety of terminal devices, we can only describe what happens in a general way. For specific details concerning individual terminals, we refer you to "Appendix III: Sign-on Procedures and Terminals."<sup>3</sup>

**Goals** The Assistant's aim is to function in a way that will be pleasant and helpful to you. To this end, the Assistant follows three general strategies.

- 1) It provides you with continuous information about its activity.<sup>4</sup>
- 2) It makes reasonable assumptions about what you want to be done when specific details are not given.<sup>5</sup>
- 3) It checks with you before carrying out a potential damaging operation.<sup>6</sup>

**Interaction with the Assistant** These strategies imply a large amount of interaction between you and the Assistant, especially when you call on it for the first time. However, you will find that interacting by means of a terminal can become tedious, particularly if the terminal is slow or if both you and the Assistant are capable of working at a much quicker pace. As you become more familiar with the Assistant, you may direct it to assume that your interaction is to be more abbreviated, just as you may at any time direct it to assume certain other things about your working environment.<sup>7</sup>

**Request Language** Requests are made to the Assistant via the terminal and are expressed in terms of a "request language." This language is designed to look very much like English and consists entirely of imperative statements. Several requests allow you to exchange information with the Assistant concerning almost anything within the scope of its knowledge.

**Behavior** At certain times, the Assistant may be attentive, which means that it is awaiting a response from you. At other times, the Assistant may be active, which means that it is trying to satisfy a request for you. Sometimes before a request is satisfied, the Assistant discovers that it needs more information from you, in which case it will ask you what it needs to know.

**Attentive Behavior** Attentive behavior is always signaled by a prompting message, which consists of two characters typed by the Assistant on the terminal. The prompting message indicates not only that the Assistant is awaiting a response from you, but also what type of response is being asked for.

**Active Behavior** When you send the Assistant a request, it becomes active and attempts to satisfy your request. It does this in three stages.<sup>8</sup>

- 1) *Verification*: The Assistant determines whether or not your request makes sense, and makes any necessary assumptions that it can when specific details are not given.
- 2) *Performance*: If the verification stage was completed successfully, the Assistant will satisfy your request. If the operation requested is at all time-consuming, the Assistant may indicate its progress at various intervals.
- 3) *Completion*: After your request has been satisfied, the Assistant indicates the final result of its actions and again becomes attentive.

**Interrupting the Assistant** While the Assistant is active, you may interrupt it at any time, causing it to become attentive again. You interrupt the Assistant in order to ask it for pertinent information or else to tell it to discontinue attempting to satisfy a request for you and to attempt to satisfy a new one.<sup>9</sup>

the left margin of the manual. (In the published version of the User's Guide, they occupy a separate column.) These keywords also give the reader a quick clue about the basic idea being presented. We are indebted to Child, Bertholle, and Beck [2] for this effective scheme.

Perhaps of most importance, the scale of the manual is small. Of course, this is reflected in the smallness of the design itself. Nevertheless, a great deal of care was exercised in eliminating details that the user should, in fact, find out for himself on the system. This is not to say that we believe the manual is incomplete or misleading; rather a great effort was made to present the system in as concise a manner as possible.

2. One of the most controversial choices we made was to present the Assistant to the user in a consciously anthropomorphic form. From the beginning we describe the Assistant as a creature, robot-like, with a goal structure, consistent behavioral rules, interactive strategies, and deductive capabilities. This idea was motivated by several considerations.

In the course of a terminal session, the user must keep track of a great deal of information. For example, the user must continually be aware of the status of his files, his current level of interaction with the system, the consequences of actions he has already taken, and the actions he may legitimately take next. Furthermore, since the actions he may perform are primitive, he must repeatedly supply redundant information over a long sequence of requests. Finally, he must constantly be on guard against destroying his own work by doing something that might seem innocuous, but results in disaster.

In most cases, the kind of information at issue here is readily available to the system. It was our intent that the Assistant take full advantage of the knowledge available to it, and relieve the user of much of the burden of constantly juggling that knowledge.

When we tried to describe a primitive knowledge-based system in a way that would be simple and nonthreatening to users, the natural step was to deliberately exploit the creature-like view which people inevitably apply to machines anyway.

However, what began as a conceptual model for the documentation quickly acquired a life of its own and repeatedly suggested consistent directions for various aspects of the design. This interaction between the documentation and the design was not limited to the conceptual model of the Assistant. In general, whenever we found a feature or concept difficult to explain clearly, we took this as a signal that the design itself was likely at fault. Whole versions of the editing requests were rejected because we could discover no simple way of explaining them.

We benefited in other ways from this view of the Assistant as a creature. For one thing, we were able to avoid much of the dryness associated with the normal type of manual. And, as the opening paragraph suggests, we were able to introduce a light touch for the reader, and thus make use of the guide a more pleasant experience.

3. Only one of the three Appendices to the User's Guide is presented in this paper. Nevertheless, it is important to note that the complete User's Guide (including all the Appendices)



comprises a document that contains everything the user has to know about the system. The reader need refer to no other documents. This is consistent with recommendations made by Vandenberg [24].

4. One of the advantages of an interactive system for users is that interaction can be used to couple the system and the user. Unfortunately, few systems provide more than negative feedback to a user, i.e., error messages. Positive reinforcement in the form of highly specific confirmatory messages or an ongoing "chatter" from the system can simultaneously teach the new user what to expect and reassure the user that what he expects is, in fact, going on. A wide body of evidence in psychological reinforcement theory supports the value of this strategy. A further benefit of this interactive strategy is that it allows the resolution of potential ambiguities that may arise in a request. Accommodating such ambiguities permits more flexibility in the language design, especially as regards abbreviated forms.

5. As Gilb and Weinberg [7] point out, extensive use of "natural" defaults is inherent in all natural language communication, and such defaults may be explicit or implicit. The Assistant is designed to take advantage of both types.

For example, if a program is to be run and no compiled versions of it is handy, the Assistant implicitly assumes that it must first be compiled. Moreover, at any time the user may explicitly direct the Assistant to make explicit assumptions about future requests. Thus, the user does not have to continue to specify file names, line boundaries, or options for requests when the system can keep track of these details.

6. The philosophy of "security checking" is not novel, but is also not commonplace, and the extent to which it is used by the Assistant may seem extreme. A frequent example is an attempt to overwrite files. Unless told otherwise, the Assistant will always inform the user that a file is about to be destroyed, ask for confirmation, and thus give the user a chance to think twice before going ahead. Another, less obvious, example is the warning a user receives when a text deletion request threatens to destroy a large part of a file. These security checks are intended to give the user confidence that the system will warn him before doing something disastrous. A skilled user may suppress most of these checks.

7. Ideally, we would like an Assistant that knows what level of detail the user needs and adapts automatically; but such intelligence is beyond the limits of cost effectiveness that we have set. The Assistant is intended to be semi-intelligent, only an incremental step toward a truly intelligent system. What we cannot accomplish with limited intelligence we have tried to accommodate with a user-driven adaptive strategy. In some ways, this is one of the least satisfying approaches that we have employed in the Assistant. Not surprisingly, the complexity of the ASSUME request, our vehicle for adaptation, reflects this.

8. There is a body of psychological evidence (see, for example, Thorndike and Rock [22])

which suggests that people "learn without awareness." One implication of these results is that the users of a computer system will infer underlying principles even if they are unaware of doing so.

The Assistant's behavioral goals are not merely "sugaring," but are accurately reflected in its responses. These goals are intended to help the user make reasonable inferences about what the Assistant will do with a particular request. For example, the first goal, verification, ensures that no request will be executed unless it makes sense semantically. In some cases, this implies that significant static pre-checking must be performed. This seems a small price to pay for relieving the user of the burden of correcting damage done by a technically legal but senseless request.

9. Most interactive systems have some form of interrupt, but like other details, the interpretation placed on it is often inconsistent or counterintuitive. The Assistant's interrupt is like a tap on the shoulder. Following an interruption, the Assistant suspends what it is doing, returns with an explanation about what is going on, and asks the user whether he wishes to continue the task. At this point, the user may reply or request additional information. If the user does request additional information, this request may itself be interrupted, but such interruption simply terminates the request for additional information and returns to the original level of interruption. Again, the user is reminded about his original interruption and is asked what to do. Thus, there is no confusing "stacking" of interrupts as, for example, in APL (Wiedmann, [26]), but interruption is always a possible and meaningful operation. This possibility of interrupting a task and carrying out a dialogue concerning the task is patterned after normal discourse (see Mann [14] and Palme, [16]). From our view, it is one of the cleanest features to emerge in our design.

10. It is a fundamental premise in the Assistant's design that users will make errors. Many interactive systems have facilities for deleting characters or lines as they are being entered. Unfortunately, a user may discover such an error well after it is made. The immediate correction feature is designed to make it simple for users to correct such errors quickly, and without retyping the entire line. If the user makes an error and, as a result, the Assistant discovers that a request is ill-formed, the Assistant will report the error. The user may then change the erroneous line with a conventional edit request, and the Assistant will automatically reissue the corrected request. While we have never seen this simple feature elsewhere, we believe that it is especially useful for lengthy editing requests and multiple request lines where typing errors are particularly frustrating.

In a similar vein, the UNDO request is a means of erasing the effect of a request that was performed but did not produce the result desired by the user. The UNDO feature will likely be limited to editing and assume requests where implementation will not cause severe difficulties.

These are both examples of the way the Assistant keeps track of things; in this case, an immediately preceding but unsatisfactory request.

**Error Conditions** There are several conditions under which a request cannot be satisfied:

- 1) the request cannot be understood or is inconsistent with what is known;
- 2) if the Assistant asks you to confirm a request and you do not comply; or
- 3) if the performance of the request fails for some reason.

25

**Immediate Error Correction** When a request cannot be satisfied, the Assistant will identify the problem and become attentive. If an error is found in the verification stage, no action will be taken. At this point, you may easily modify and reissue your request using a request correction facility. Alternatively, you may issue an entirely new request.<sup>10</sup>

**Files** A file is a named collection of information that the Assistant maintains for you. The Assistant's primary function is to provide you with a means of creating, manipulating, and performing various operations with files. Most files that you will use will be files of text, and many of these will be PASCAL programs. When you create a file, you give it a name. From then on, both you and the Assistant refer to that file by the file's name.<sup>11</sup>

**Preserving Files** Any file that you create during a session with the Assistant will be kept for the duration of the session. It may be kept for future sessions provided that you specifically ask the Assistant to preserve it for you. No file will be discarded without your prior approval. Files previously preserved can be modified at any time. However, at some point the Assistant must be told whether or not these modifications are to be preserved as well. Once a modified file has been preserved, its previous condition is lost forever.<sup>12</sup>

**Assumptions** The Assistant retains information about what you have done and what you have explicitly asked it to assume. Initially, the Assistant uses some basic assumptions about how you, as a beginner, would want it to behave.<sup>13</sup> Assumptions, as we have said, enable the Assistant to reach reasonable conclusions about what you want done when certain details are omitted from a request. Thus, the use of assumptions frees you from having to supply excruciating amounts of detail.

**Limitations** As we stated earlier, the Assistant has a very limited understanding. It can make only very simple deductions based on its restricted knowledge. When you try to give it a request that it does not understand, it will tell you so, but it cannot really inform you of the limits of its own intelligence. This does not mean that its intelligence is illusory. In fact, you may very well find its perceptiveness surprising at times.<sup>14</sup>

#### SOME NOTATION<sup>15</sup>

**Grammatical Notation** Every language has a grammar, and the Assistant's request language is no exception. Because the Assistant identifies your requests by their form, grammar is especially important in communicating with it. In the descriptions that follow, we employ a special notation to describe the grammatical form of each request. The rules for this notation are as follows.

1) **Keywords:** Words shown in uppercase are *keywords*. Keywords are like guideposts to the Assistant. They signal what to do and what to expect. Except for PRESERVE, RESTORE, and DESTROY, any keyword may be abbreviated by its first letter. If not abbreviated, a keyword must be spelled out correctly (see note 17).

2) **Objects:** Words shown in lowercase and connected with hyphens ("-") are names for the objects of a request that you supply to make the request specific, such as the name of a file, a mode of interaction or a piece of text.

3) **Alternatives:** Keywords or objects that are grouped together and separated by slashes ("/") are mutually exclusive alternatives. For example,

11. While a serious attempt was made in the Assistant's design to avoid the terminology of conventional systems, the concept of a file seemed inevitable. In a private correspondence, Hoare [8] suggested the alternative notion of "books" or "folders" supported by an appropriate graphic display. Our decision to support printing terminals ruled this out.

12. The Assistant uses a simple two-level file system. A good deal of effort went into designing this system so that its operation is largely automatic and transparent to the user. When the user directly refers to a new file, a current temporary copy of it is created. All operations are performed on the current version. At the end of an interactive session, the Assistant asks the user what to do with files that do not have equivalent permanent copies. Although the user must be aware that, potentially, there are two copies of his file, the management of these files is left largely to the Assistant. Specific file manipulation requests enable a user to preserve the current version of a file or restore it to its previously preserved condition.

In retrospect, it is somewhat surprising how much time we spent designing this scheme. Yet, we believe that the concepts of file restoration and preservation in the Assistant are unusually simple.

13. The assumptions for beginners take nothing for granted and attempt to assure that no beginner will be lost too easily.

14. We had reservations about presenting the Assistant as a semi-intelligent creature with moderate self-consciousness that understands a narrow natural-language subset. There is always the danger that naive users will come to expect too much and thus be frustrated. We have tried to compensate for this by emphasizing limits, but it may not be sufficient. Nevertheless, we feel that the benefits of an approachable conceptual framework for people are significantly greater than the problems it may create.

15. Despite our desire to keep notation and terminology to a minimum, we felt compelled to resort to a kind of context-free grammar. Notations, even simple context-free grammars, can at first be difficult for many users. We attempt a gentle introduction to the use of a few grammatical notations. It is likely that this complexity of the documentation reveals what is probably a weakness in design.

n/ALL

means that either "n" (a number) or the keyword "ALL" may be specified, but not both.

4) *Options*: Keywords, objects, or any groupings of these that are in parentheses represent parts of a request whose use is optional. For example,

QUIT (QUICKLY)

means that you may say "QUIT QUICKLY" or simply "QUIT."

5) *Ordering*: Keywords, objects, or groupings of these may only be specified in the order in which they appear in a rule.

### REQUEST LANGUAGE SUMMARY<sup>16,17</sup>

#### General Requests

EXPLAIN	(name)
SHOW	(name)
ASSUME	assumption
GRIPE	
UNDO	
QUIT	(QUICKLY)

#### Editing Requests

NEXT	(lines-of-text)	
PREVIOUS	(lines-of-text)	
LIST	(lines-of-text)	
DELETE	(lines-of-text)	
TRANSFER	(lines-of-text)	INTO file <sup>18</sup>
COPY	(lines-of-text)	INTO file
INSERT	(new-lines-of-text)	(BEFORE/AFTER/OVER)
MAKE	text	new-text

#### File Requests

PRESERVE	(file-list)
RESTORE	(file-list)
DESTROY	(file-list)

#### Program Requests

RUN	(file-list)	(WITH parameter-file-list)
VERIFY	(file)	(INTO file)
FORMAT	(file)	(INTO file)
BIND	(file-list)	INTO file

### GENERAL REQUESTS

The information requests EXPLAIN, SHOW, and ASSUME provide you with the means of exchanging information with the Assistant. You may direct the Assistant to make assumptions about your environment or you may ask it for information about current assumptions, requests, the request grammar, and so on. The use of these requests should make it unnecessary to refer to this User's Guide while interacting with the Assistant.

**Explaining Concepts** The EXPLAIN request is your means of getting general information in order to understand something about the Assistant that is not clear. In order to ask about something just say

EXPLAIN (name)

The "name" you give can be any one of a number of words associated with the

26 16. A major concern in the design was to limit the scale of the Assistant. This was one of the most difficult issues to confront. The tendency to expand and enlarge, to add "powerful" and "important" features was overwhelming. As Miller [29] pointed out in a stimulating but inconclusive paper, "The Magical Number Seven Plus or Minus Two," there definitely seem to be small limits on our capacity for dealing with large numbers of conceptual objects, but these limits are extended by a phenomenon known as "chunking," in which aggregates can be formed. In spite of the chunking phenomenon, we believe there is a strong intuitive case to be made for keeping things small.

A common criticism voiced over the Assistant's design is that it is a "toy." This was certainly not our intent. However, we have rigorously excluded any feature which we felt would be of use to only a small fraction of users. We believe that the Assistant is an uncommonly simple solution to providing a pleasant and productive working environment for a majority of programmers.

From the request language summary the small scale and symmetry of the Assistant are immediately apparent. What is not so apparent is the capability that lies within this simplicity. Users of HOPE, our prototype of the Assistant's editing requests, have been surprised by the power of what they took to be a fairly simple-minded editor.

17. Another major design decision we made was to base the Assistant's request language on a limited English phrase structure. There were a number of reasons for this choice. The natural language of interaction between people is natural language. Even individuals exceptionally experienced with notation have still greater training in natural language. Thus, our aim was to exploit this natural language experience.

Because a reasonable body of experimental data (see, for example, Weist and Dolezal [30] and Epstein and Arlinsky [27]) suggests that people have difficulty in manipulating language-like information that violates normal syntactic structure, we tried to follow normal syntax as closely as possible; and we tried to choose the shortest, most apt, and most orthogonal set of keywords. Short words were chosen not out of typing considerations, but because they occur more frequently and are easier to recall.

While we tried to copy English grammar closely, we did not allow the meaningful reordering of phrases permitted in English, e.g., "into A, copy B." We avoided this because of the ambiguities it might introduce into the request language, especially in its abbreviated form. It seems more desirable now to use a more relaxed syntax and resolve ambiguities with an interactive exchange with the user.

Seemingly at odds with the decision to follow natural language syntax strictly was the requirement that the request language have an effective abbreviated form. The ideal, of course, would be to have special function keys for each word, but the real world of ordinary terminals precludes that.

The solution was to introduce the uniform abbreviation rule that *any* keyword can be abbreviated by its *first* letter. Furthermore, abbreviated keyword sequences can be typed *without* intervening spaces. These two rules result in an abbreviated form of requests that

request language, error conditions, the Assistant itself, or various concepts behind it, like assumptions or files.

If you say EXPLAIN, omitting any name, the Assistant will respond by giving you information concerning the last thing that you have done or that has happened to you. Each time you say "EXPLAIN" the Assistant will provide you with more information concerning the topic at hand. In addition to its explanation of the given topic, the Assistant may refer you to other related topics.

If the Assistant does not have information on a given name, it will tell you so. If all its information is exhausted, the Assistant will, if possible, suggest external sources (consultants, references, etc.) that you might seek out.<sup>19</sup>

**Getting Examples or Specific Data** When you want examples of the request language or specific data concerning files or your working environment, say

SHOW (name)<sup>20</sup>

In addition to the normal names of things you might ask about, there are several words which will direct the Assistant to show you some special things. These are

TIME	The current time of day.
ASSUMPTIONS	All of your current assumptions.
FILES	The names and information concerning your currently preserved files.

**Giving Assumptions** In order to tell the Assistant to make specific assumptions about your environment, say

ASSUME assumption

Assumptions fall into several categories. You can specify one of two modes of interaction by saying

ASSUME INTERACTION IS TERSE/LONG<sup>21</sup>

These two modes are interpreted as follows:

TERSE	Gives highly abbreviated messages or none at all. Intended for the hotshot user.
LONG	Gives loquacious messages, spelling everything out from A to Z. Intended for the naive or inexperienced user.

Another category determines the amount of interaction you want the Assistant to assume regarding security checks for potentially dangerous operations. You can specify how much security you want by saying

ASSUME SECURITY IS CAUTIOUS/RISKY<sup>22</sup>

Other uses of the ASSUME request are given further on.

**Complaints, Comments, and Suggestions** The Assistant, via EXPLAIN and SHOW, is designed to help you as much as possible within its limited knowledge. However, sometimes this is not enough. You cannot really tell the Assistant your problems and get any kind of sympathy or advice from it. You can, however, tell the people in charge your problems through the Assistant by saying

GRIPE

The Assistant will then go into a special attentive mode where you may type in a message of any number of lines. You leave this special mode of interaction by interrupting the Assistant and making a new request. The text you type will be stored, and at regular intervals all the messages sent by you and others will be sifted out and examined by the people responsible for maintaining the Assistant.<sup>23</sup>

is fast and easy to type. Because the rule is so simple, the user can think in the long form while typing its abbreviation. (Because of their potential danger to the user, the three file requests were excluded from this general rule and cannot be abbreviated. This now seems paradoxically inconsistent.)

A variety of data suggest the first letter abbreviation rule. A paper by Freedman and Landauer [6] points to the usefulness of the initial letter as a recall clue, and some recent work by Chin-Chance [3] indicates the importance of the initial letter (for adults at least) as a discrimination cue.

This approach to abbreviations is not a "minor" issue. One of the least thought out philosophies of almost every system we have seen is its abbreviation strategy. Abbreviations, like other so-called "details" of design are often very critical, for such details may be the most frequently encountered features of a system. From the user's point of view, ours is a powerful convention. From a designer's point of view, this convention was almost impossible to live with. On many nights we took a thesaurus to bed.

An argument commonly advanced against our abbreviation rule has been that we could not easily expand the keyword list, i.e., add new requests. In rebuttal, we suggest that such additions would be best accomplished by a complete redesign, if all the interlocking design aspects are to receive the consideration they deserve. Furthermore, as the ASSUME demonstrates, any keywords within a request are free from conflict from keywords within other requests.

18. The TRANSER and COPY requests are good examples of our attempts to follow a limited English phrase structure. Rather than use conventional notations like "TRANSFER, lines-of-text, file", we borrow from natural English phrase structure.

Unfortunately, we were not completely successful in following English grammar. From a grammatical point of view, the MAKE request would be better as "CHANGE text TO new-text." However, this suffers from the defect of requiring two levels of delimiting, the string delimiters that bracket text, and the syntactical delimiter "TO." Of course, all of this results from the clash between notation (string delimiters) and natural language, an impossible dilemma.

19. A number of interactive systems now incorporate on-line assistance features (e.g., see Teitelman [21]). To the best of our knowledge none of these are integrated into the system so as to take advantage of an awareness of what is going on. The idea of an integrated assistance feature follows naturally from the general interactive strategy of the Assistant and, as such, is simply a request from the user for greater amplification.

The benefits of this approach are several. The user can directly get information that in a conventional system would only be available in a reference manual. Furthermore, this information can be specialized to his situation. Finally, this information is provided in the context of an actual circumstance where its teaching value and reinforcement potential is greatest.

**One Last Chance** If you make a request and you wish you had not, you may undo the effect of that request by saying

UNDO

The effects of the most recent request made are cancelled, and you may then proceed as if nothing had ever happened.

**Leaving the Assistant** In order to dismiss the Assistant say

QUIT (QUICKLY)

Before the Assistant will let you go, it will tell you what files have been created or changed and are still to be preserved, and ask you which of those you wish to keep. Furthermore, it will ask you whether or not you want to preserve any new assumptions that you have given it. Finally, it will make doubly sure that you wish to leave before it will let you go.

If you add "QUICKLY" to the request, it will assume that you have already preserved everything you want to keep and will let you go without any fuss.<sup>24</sup>

### EDITING REQUESTS

Text editing is a process of creating, maintaining, and updating files of text (such as programs, data files, chain letters, or what have you). The Assistant's editing requests make it possible to insert, delete, and substitute text to change the layout and spacing of text, and even to move blocks of text from one file to another.<sup>25</sup>

Editing text commonly requires that a number of changes be made to a particular file. Rather than repeatedly specifying the file to be edited in each request, the Assistant always assumes you want to edit the currently assumed file. (For an explanation of the "currently assumed file" and how it works, see "File Requests-File Assumptions.")

**The Current Line** In making editing requests, you must always have some means of specifying what it is you want changed. The Assistant always assumes that a request is made relative to a "current line." Initially, the current line is the first line of the file. Thereafter, each request that references specific lines causes the last line referenced to become the new current line.<sup>26</sup>

**Specifying Text** Editing may be performed on whole lines or on pieces of text within a line. Operations on whose lines may be specified by giving the number of lines from the current line or by giving a piece of text which appears on a line. References to pieces of text require a special notation to describe the text. This notation has the form

=text=

The given "text" is any actual sequence of characters. The symbol "=" represents any special character which is neither a letter, digit, space, or semicolon (";"). This special character is used to "bracket" the actual character sequence. Since this character indicates both the beginning and ending of the desired text, it must be a character which does not appear in the text itself.<sup>27</sup>

An example editing session is shown in Fig. 1 at the end of this section.

**Moving Forward** To move the current line forward say

NEXT (lines-of-text)

There are several ways of describing how many lines of text to advance. The NEXT request has the following variations.

1) NEXT

The Assistant moves the current line forward one line.

20. The SHOW request is also meant to provide pedagogical examples of the request language. For example, if the user types "SHOW MAKE," the Assistant will give examples of the use of the MAKE request. For both the EXPLAIN and SHOW requests, it is expected that over time more information will be added to the Assistant's knowledge base. Coupled with the GRIPE request, this seems to be a viable approach for improving the Assistant's behavior as our knowledge of what needs explanation expands.

21. Our original design was based on three modes of interaction, TERSE, MODERATE, and LONG. We are grateful to Houe [8] for pointing out that with a good implementation of the EXPLAIN request two modes should be sufficient.

22. As Gilb and Weinberg [7] point out, at times and for some users, automatic protection and forced interaction may be a nuisance.

23. The importance of long range user feedback in maintaining a system cannot be underestimated. In providing a specific request for this, we emphasize its importance and make spontaneous complaints possible. Furthermore, we can take immediate advantage of the system itself to capture inside information about the current state of affairs, which may help us in interpreting a user's complaint.

24. The QUIT request is a good example of our desire to make reasonable and safe assumptions about the user's behavior and still allow more skilled users to override these assumptions.

25. In most systems, editing must take place in a special mode or environment. These systems require users to shift levels. The requirement that editing languages be terse usually conflicts with the large scale of the rest of the system. A special editing environment is the logical, if cumbersome, solution to this problem. Then again, many editors are built as independent subsystems and only later incorporated into the main system.

Various studies (e.g., see Turner [23] and Boies [1]) have shown that editing usually accounts for better than 50 percent of the average interactive system's work. Furthermore, the nature of the program development process often leads a user to ping-pong between editing and other tasks.

For these reasons, we believe that a text editor must be designed to be an integral part of an interactive programming environment. Central to this belief is our feeling that a user should have access to all the capabilities of the system while editing and vice versa. The use of the "assumed" or default file together with the small scale of the Assistant enable us to keep a single-level system for all requests. We are indebted to Stemple [19] for making the strong case for this.

26. One of the larger and more difficult decisions we made was to orient the editing requests of the Assistant around the concept of a "current line." Some editors are "character based" (that is, the user's position in the file may occur in the middle of a line), and others are page oriented (i.e., the interaction is always in terms of multiple lines of text).

## 2) NEXT n

The Assistant moves the current line forward n (where n is a number) lines.

## 3) NEXT ALL

The Assistant moves the current line forward to the *last* line in the file.

## 4) NEXT =text=

The Assistant moves the current line forward to the next line containing an occurrence of the specified text.

## 5) NEXT n =text=

The Assistant moves the current line forward to the "nth" line containing an occurrence of the specified text.<sup>28</sup>

## 6) NEXT ALL =text=

The Assistant moves the current line forward to the *last* line containing an occurrence of the specified text.

In all of the editing requests, "lines-of-text" has the same general form as shown above.<sup>29</sup>

**Moving Backward** To move the current line backward say

## PREVIOUS (lines-of-text)

The PREVIOUS request is exactly the reverse of the NEXT request. Note that PREVIOUS ALL takes you to the first line in the file.

**Displaying Lines** To display one or more lines of text just say

## LIST (lines-of-text)

The variations on the LIST request are similar to the NEXT and PREVIOUS requests.

## 1) LIST

Only the current line is displayed on the terminal.

## 2) LIST n/ALL

The Assistant displays the next n (or ALL) lines including the current line.

## 3) LIST n/ALL =text=

The next n (or ALL) lines containing the specified text are displayed.

**Deleting Lines** In order to delete one or more lines of text you say

## DELETE (lines-of-text)

This operation is virtually identical to the LIST request with the difference that the particular lines specified are not displayed but *removed* from the assumed file.<sup>30</sup>

**Moving Lines** To move one or more lines of text out of the assumed file and into another file say

## TRANSFER (lines-of-text) INTO file

This request removes the lines of text you specify from the assumed file and puts them into the other file that you name. The lines that are removed will replace the previous contents of the file.<sup>31</sup>

29

Obviously, the kind of terminals in use and the kind of terminals in use and the kind of text to be edited enter into this decision. We made a deliberate design decision to orient the Assistant around moderate speed (10-30 eps), typewriter-based terminals without a graphic display facility, as these are at present the most commonly used. We also concentrated primarily on the problem of editing programs. While we did not rule out the possibility that the editor might be used for ordinary language text, the special problems of editing such text were not addressed (see Miller and Thomas [15]).

It might have been better to design the Assistant for a more advanced type of high-speed terminal. Indeed, with a bit more storage, some of the "intelligent" terminals made possible by recent advances in semiconductor technology seem entirely capable of supporting an Assistant locally. The parallelism of display, cursor facilities, definable function keys, and the fast display rate afforded by such terminals would make possible substantial improvements in the design of the Assistant, particularly with regard to the editing requests and the management of defaults.

In our opinion, most editors based on the "current line" concept suffer from the drawback that the user must mentally keep track of what the current line is. This defect results from an inconsistent strategy with respect to line pointer movement. In the Assistant we have deliberately avoided this possible confusion.

The current line is *always* the last line seen by the user. The advantage of this strategy is that the terminal is always displaying the current line. The disadvantage is that the examination of text may force an extra step, i.e., moving back to the beginning of text which is to be displayed. Clearly, there are arguments on both sides. Here again, we believe that the value of the general rule outweighs the merits of a special case. Certainly, this issue deserves some thoughtful experiments.

27. Here again, our use of special notation reveals a weakness of design. We remain dissatisfied with this, but find other alternatives even less attractive.

28. A particularly sticky, but important, detail. Should it be the *n*th occurrence or *n*th line containing an occurrence? The former seems right for a character-oriented editor, while the latter seems more suited to our line-oriented editor. Endless hours were spent on this issue, with no clear resolution.

29. Getting all the editing requests to conform to the same general format for target text patterns was the result of great attention to detail and numerous debates about the proper function of requests. In doing so, we significantly reduced the amount of information a user must learn and remember.

30. An intended security check confirms major deletions with the user.

31. It is not obvious from the User's Guide, but the TRANSFER request is not only intended to excise lines from a file but is the basic mechanism for moving blocks of text within a file. By transferring lines of text to a

TEXT EDITING SESSION: The user wishes to edit an existing file called POEM.  
 (← indicates the current line)

32

```
--ASSUME CURRENTFILE IS POEM
What am I? ←
--LIST ALL
What am I?

They choose me from my brothers: "That's the
actual number of lines
Nicest one," they said,
Candle in my head;
And they carved me out a face and put a
Night was dark and will
But when they lit the fuse, then I jumped! ←
--PREVIOUS /actual
actual number of lines ←
--DELETE 1
Nicest one," they said, ←
--NEXT 1
Candle in my head; ←
--TRANSFER 1 INTO HOLD-FILE
and they carved me out a face and put a ←
--INSERT HOLD-FILE AFTER
Candle in my head; ←
--ASSUME MARGIN IS $
--ASSUME ELLIPSIS IS ...
--INSERT /$$
←
--INSERT
↑↑And they set me on the doorstep. Oh, the
↑↑
And they set me on the doorstep. Oh, the ←
--NEXT 1
Night was dark and will ←
--MAKE /will/wilpqrs
Night was dark and wilpqrs ←
--UNDO
Night was dark and will ←
--MAKE /will/wild;
Night was dark and wild; ←
--NEXT 1
But when they lit the fuse, then I jumped! ←
--MAKE /fuse..jumped!/candle, then I $$$smiled!$
Smiled! ←
--PREVIOUS ALL
What am I? ←
--LIST ALL
What am I?

They choose me from my brothers: "That's the
Nicest one," they said,
And they carved me out a face and put a
Candle in my head;

And they set me on the doorstep. Oh, the
Night was dark and wild;
But when they lit the candle, then I
Smiled! ←
--PRESERVE POEM
```

```
**The current line is the first line of POEM.
**The entire file is displayed.
**This is one blank line.

**The current line is moved backward.
**The current line is deleted and the line
**following becomes the new current line.
**The current line is advanced one line.
**HOLD-FILE contains "Candle in my head;"
**The contents of HOLD-FILE are inserted
**after the current line.
**A blank line is inserted and becomes
**the current line.
**New lines are requested.
**The Assistant is interrupted.
**The last line inserted is now the
**current line.
**One word is changed (incorrectly).
**The previous request is undone.
**The word is changed again (correctly
**this time).
**The last line is altered and another line
**is added by using the MARGIN symbol.
**The current line is moved back to the first
**line in POEM, and the entire file is listed.

**The new version of POEM is preserved.
```

## FILE REQUESTS

33

**Preserving Files** Files are normally preserved only during the dialogue at the end of your terminal session. However, if you are wary of erratic behavior on the part of the Assistant or do not feel at all confident of reaching the end of your session, then you may explicitly preserve files at any time by saying<sup>40</sup>

**PRESERVE** (file-list)

If any of the files named in the file list do not exist or have not been changed since last preserved, then no action will be taken. You should either correct the request or enter a new request. (See "Robot's Rules of Order--Immediate Request Correction.")

**Restoring Files** If any files that have been previously preserved are changed in any undesirable way, then you always have the recourse to restore those files to their most recently preserved condition by simply saying

**RESTORE** (file-list)

If any of the files in the file list have not been previously preserved or if any of them have not been changed since they were last preserved, then none of them are restored, and you should proceed as above to correct or reissue the request.

**Destroying Files** If you no longer wish to keep a preserved file or if you run out of storage space and must discard some files, then you may completely and permanently annihilate any file by saying:

**DESTROY** (file-list)

Beware. Once a file is destroyed, there is no way of getting it back very easily. Spare yourself some agony and make sure that you want a file destroyed before you destroy it.<sup>41</sup>

**File Assumption** All of the editing requests in the previous section depend on having a "currently assumed" file to edit. In order to specify what file is to be assumed simply say

**ASSUME CURRENT FILE IS** file

Except where noted, all other requests use the assumed file if a file is not given explicitly.

**File Renaming** In order to change the name of the **CURRENTFILE**, all you have to say is

**ASSUME NEWNAME IS** new-file-name

## PROGRAM REQUESTS

**Executing Programs** In order to execute a PASCAL program say<sup>42</sup>

**RUN** (file-list) (WITH parameter-file-list)

The "parameter-file-list" is a list of file names that are to be substituted for the formal file parameters in the program header of your PASCAL program. If a file exists in your program header but is omitted from your parameter-file-list, then the file name assumed is that of the formal parameter in the program header. For example, if your program header is

**PROGRAM DUMMY (FILE1, FILE2, FILE3, FILE4, FILE5);**

and you type

**RUN DUMMY WITH XYZ, ABC, DEF**

39. We believe that users learning a complex task (for example, a new computer system or a new natural language) are helped by examples. This page of the Assistant's manual gives an example of an entire user dialogue. We believe that even this example is not really sufficient for proper understanding of the Assistant's editing behavior, and that the User's Guide as a whole should probably be more example-based.

40. **PRESERVE** also provides the user with a defense against an unreliable environment. However, if a system is subject to frequent crashes and the user must frequently interrupt his dialogue to save his work, the result will be a considerable waste of both the computer's and the user's time. Thus, reliability is also a significant human engineering concern.

41. A secondary protection feature that might make this warning unnecessary would be the automatic archiving (for a time) of every file to be destroyed. This was one of the few instances in which implementation considerations were allowed to restrict the design. The archiving of destroyed files now seems to be a less formidable requirement.

42. The spirit of the **RUN** request is that it runs a PASCAL program. The form that the program is in is irrelevant. If need be, the program will be compiled, but this is transparent to the user unless errors are found. The mechanics of keeping track of source and object versions if they are distinct is managed automatically by the Assistant. Of course, complete control of the computer passes to the user's program and the Assistant disappears. From our point of view, this is bad; but the alternative, incorporating a kernel Assistant into the run-time program, seemed overwhelming. Building a kernel of the Assistant into the PASCAL run-time system now seems inescapable, despite the implementation difficulties.



then your request will be interpreted as

```
RUN DUMMY WITH XYZ, FILE2, ABC, DEF, FILE5
```

34

If more than one file name is given in the file list, the first file named is assumed to contain the main program segment, and all the others to contain external procedures. For further information on the linking of externals to PASCAL programs, see "Appendix II: Linking External Procedures to PASCAL Programs."

If a PASCAL error exists in your program, you will be told so, and your program will not be executed. To see a listing of those errors use the VERIFY request described below.

**Verifying Programs** In order to get a summary of errors in your PASCAL program just say

```
VERIFY (file) (INTO file)
```

Depending on whether you have assumed a TERSE or LONG mode of interaction, you will get either a brief summary of error messages or a full listing of your program with error messages. If "INTO file" is specified, the verification will be put into the file instead of being displayed at the terminal.<sup>43</sup>

**Formatting Programs** In order to format your PASCAL program according to standard pretty-printing conventions say

```
FORMAT (file) (INTO file)
```

If "INTO FILE" is specified, the results of the format will be put into that file; otherwise, they will be displayed at the terminal.

The FORMAT request takes a text file containing a PASCAL source program and reformats it according to a set of standard spacing conventions. FORMAT in no way affects the logical ordering of the program; it merely rearranges the file into a standard format. The standards have been developed so that the reformatted program is aesthetically appealing, logically structured, and above all, readable.

Extra spaces and extra blank lines found in the text are kept. You may improve the readability of your program even more by adding extra spaces and blank lines beyond those inserted by the Assistant.<sup>44</sup>

For example, if your current file looks as follows:

```
TYPE SCALE = (CENTIGRADE, FAHRENHEIT);
FUNCTION CONVERT ((* FROM *) DEGREES: INTEGER;
(* TO *) NEWSCALE: SCALE): INTEGER;
BEGIN IF (NEWSCALE = CENTIGRADE) THEN
CONVERT := ROUND((9/5*DEGREES) + 32) ELSE
CONVERT := ROUND(5/9*(DEGREES - 32)) END;
```

and you then type "FORMAT," the reformatted program will be printed at your terminal as follows:

```
TYPE SCALE = (CENTIGRADE, FAHRENHEIT);
FUNCTION CONVERT ((* FROM *) DEGREES: INTEGER;
(* TO *) NEWSCALE: SCALE): INTEGER;
BEGIN
  IF (NEWSCALE = CENTIGRADE)
  THEN
    CONVERT := ROUND ((9/5*DEGREES) + 32)
  ELSE
    CONVERT := ROUND (5/9*DEGREES - 32))
END;
```

**Binding Programs** There may come a time when you simply will not be modifying a program any further, but executing it very often. For execution efficiency

43. Complementary to RUN, the VERIFY request is strictly for checking a program. Object code might be generated but that is the Assistant's business, not the user's.

44. The FORMAT request is based on a program that automatically prettyprints PASCAL text. A detailed description of this program appears in Hueras and Ledgard [10]. This program contains several features that we believe are unique. For one, the program needs no information from the user other than the file itself. For another, the program handles even program fragments. Our initial feeling was that developing an automatic formatting program was easy. This did not turn out to be the case.

you may bind you program into an execute-only file by saying

```
BIND (file-list) INTO file45
```

If there are any PASCAL errors in any of your programs, the programs will not be bound and you will be informed of your situation.

### ROBOT'S RULES OF ORDER

1) **Prompting Messages** Two prompting characters are always printed by the Assistant to indicate its attentiveness. The characters indicate what type of response is expected from you.

Prompting Characters	Response Type
<i>(Note: "␣" signifies a space.)</i>	
-- <sup>46</sup>	Requests
++	New-Text Input
//	Caution Checks
?␣	PASCAL Program Input

2) **Information Requests** An information request may be issued whenever the Assistant is attentive, regardless of what prompting message has been given. The only exception is the "?␣" prompt, which is issued by a PASCAL program, *not* the Assistant.<sup>47</sup>

3) **File Names** File names may be of arbitrary length, but no less than two characters. The characters that may be used are letters, digits, and the break character "-". The first character must be a letter and the last cannot be the break character.<sup>48</sup>

For example,

```
SQUARE-ROOT-PROGRAM
SINE-COSINE-FUNCTION
```

are legitimate file names, while names such as

```
3QX (does not begin with a letter)
A (contains too few characters)
H3.1 (contains an illegal character)
```

are not.

4) **Abbreviations and Request Spacing** All words in requests, with the exception of file names and the request names PRESERVE, RESTORE, and DESTROY may be abbreviated by their first letter.<sup>49</sup> Spaces in a request may be omitted, with the exception that files and file lists must be preceded and followed by a space.<sup>50</sup>

For example,

```
TRANSFER 3 INTO ALPHA
NEXT 5
```

may be abbreviated as:

```
T3I ALPHA
N5
```

5) **Multiple Requests** You may type in more than one request on a line any time by separating each request by a semicolon (";").<sup>51</sup>

6) **Interaction Control** Each of the words TERSE, LONG, CAUTIOUS, or RISKY may be appended to any request on a line to temporarily override the currently assumed mode of interaction for the duration of the request.<sup>52</sup> For

35

45. A vestigial concession to manual program management strategies. On a high level architecture like the Burroughs B7500 it would be irrelevant. (We may have been shortsighted as it now seems to us that even a conventional loader environment could probably be effectively managed automatically by the Assistant.)

46. Another detail. We spent a lot of time trying to choose meaningful and distinctive graphics for these prompting symbols because they will be seen so frequently.

47. Probably our darkest hour.

48. The break character for compound names in natural language is the hyphen. Thus, for the request language we use the hyphen to connect compound names. We believe this convention is easy to use and well-founded.

49. There are a number of two-word keywords, like CURRENTFILE, in the request language. It is certainly not clear how to abbreviate them.

50. The requirement that spaces delimit file-names was intended to eliminate ambiguity. Ambiguity now seems rare enough to be worth tolerating.

51. Allowing multiple requests on a line enables the more experienced user to build compound requests. In an environment with slow reaction time it may give the user more satisfaction to work with longer request lines and adapt to the slower pace. As Palme [16] and others have pointed out, such adaptation is comparable to the adaptation that takes place in natural human dialogue.

This feature is not novel, but the Assistant's interactive "chatter" during execution of a request line and the immediate request correction facility make it more effective.

52. There is some doubt in our minds as to the value of this.

example, if you are currently assuming LONG messages but would rather not see a LONG message for an EXPLAIN request, then you would type

36

EXPLAIN (name) TERSE

7) **Immediate Request Correction** Whenever a request is given and not satisfied due to an error, you may correct the error by modifying the request, rather than retyping it entirely. To do so, simply type

=old-text=new-text=

In this case, "new-text" will replace the first occurrence of "old-text" found in the erroneous request, and the Assistant will then automatically attempt to satisfy the request again for you. If old-text is not found in the erroneous request, then nothing is done, but you still have the option of trying to modify the request once more. "=" may be replaced by any character other than a letter, or ";", which is neither in old-text or new-text. It is used simply as a separator and is not considered part of either old-text or new-text.

---

APPENDIX I  
THE ASSISTANT AT A GLANCE

*General Requests:*

EXPLAIN	(name)		
SHOW	(name)		
ASSUME	INTERACTION	IS	TERSE / LONG
	SECURITY	IS	CAUTIOUS / RISKY
	CURRENTFILE	IS	file
	NEWNAME	IS	new-file-name
	MARGIN	IS	special-symbol / NULL
	ELLIPSIS	IS	special-symbol / NULL
	JOKER	IS	special-symbol / NULL
	UPPERLIMIT	IS	CURRENTLINE / NULL
	LOWERLIMIT	IS	CURRENTLINE / NULL
GRIBE			
UNDO			
QUIT	(QUICKLY)		

*Editing Requests:*

NEXT	(n/ALL)	(=text=)	
PREVIOUS	(n/ALL)	(=text=)	
LIST	(n/ALL)	(=text=)	
DELETE	(n/ALL)	(=text=)	
TRANSFER	(n/ALL)	(=text=)	INTO file
COPY	(n/ALL)	(=text=)	INTO file
INSERT	(=text= / file)		(BEFORE / AFTER / OVER)
MAKE	(n/ALL)	=text=	=new-text=

*File Requests:*

PRESERVE	(file-list)
RESTORE	(file-list)
DESTROY	(file-list)

*Program Requests:*

RUN	(file-list)	37	(WITH parameter-file-list)
VERIFY	(file)		(INTO file)
FORMAT	(file)		(INTO file)
BIND	(file-list)		INTO file

*Request Modifiers:*

TERSE / LONG  
CAUTIOUS / RISKY

*Request Correction:*

=old-text=new-text=

*Request Spacing:*

**Requests** Spaces in a request may be omitted, with the exception that files and file-lists must be preceded and followed by a space.

**File-names** A file-name must be comprised of at least two characters. Characters that may be used are letters, digits, and the break character (" "). The first character of a file-name must be a letter, and the last character cannot be a break character.

**File-lists** A file-list is a list of file-names separated by commas (",").

**Multiple Requests** More than one request may be typed on a line provided that each request is separated by a semicolon (";").

*Prompting Characters and Response Types:*

-- Requests  
++ New Text Input  
// Caution Checks  
?# PASCAL Program Input

*Conventions:*

- 1) Uppercase letters denote reserved keywords.
- 2) Lowercase letters denote objects.
- 3) Parentheses denote optional keywords or objects.
- 4) A slash ("/") denotes mutually exclusive alternatives.
- 5) " " denotes a space.
- 6) All keywords may be abbreviated by their first letter, except for PRESERVE, RESTORE, and DESTROY.

**ACKNOWLEDGMENT**

We would like to express our appreciation to Dr. C. Wogrin and the University of Massachusetts Computing Center for their generous support and interest in this project.

Many students have also contributed to this work. D. Winters, R. Chow, P. Haynes, G. Madelung, and D. Tarabar helped design and build early prototypes of major system components. The students of COINS 790T and 7900 have all contributed to the ideas in this work.

Finally, our thanks to M. Marcotty, who has continually stimulated our efforts to develop a pleasant environment for users.

We are grateful to the National Science Foundation and the U.S. Army Research Office for their support of this effort. Special thanks are due to J. B. Martin and the Publications Staff at IEEE Headquarters for their tasteful assistance with the typesetting of this paper.

## REFERENCES

- [1] S. J. Boies, "User behavior on an interactive system," *IBM Syst. J.*, no. 1, pp. 2-18, 1974.
- [2] J. Child, L. Bertholle, and S. Beck, *Mastering the Art of French Cooking*, vol. 1. New York: Knopf, 1961.
- [3] S. A. Chin-Chance, "A mathematical model of word recognition strategies," Ph.D. dissertation, Dep. Educational Psychol., Univ. Hawaii, Honolulu, 1978.
- [4] J. E. Cooke and R. B. Bunt, "Human error in programming: The need to study the individual programmer," Dep. Comput. Sci., Univ. Saskatchewan, Canada, Tech. Rep. 75-3, 1975.
- [5] C. A. Cuadra, "On-line systems: Promise and pitfalls," *J. Amer. Soc. Inform. Sci.*, Mar.-April, 1971.
- [6] J. L. Freedman and T. K. Landauer, "Retrieval of long-term memory: Tip-of-the tongue phenomenon," *Psychol. Sci.*, vol. 4, no. 8, pp. 309-310, 1966.
- [7] T. Gilb and G. Weinberg, *Humanized Input: Techniques for Reliable Keyed Input*. Cambridge, MA: Winthrop, 1977.
- [8] C.A.R. Hoare, private communication, 1976.
- [9] H. O. Holt and F. L. Stevenson, "Human performance considerations in complex systems," *Science*, vol. 195, pp. 1205-1209, 1977.
- [10] J. Hueras and H. Ledgard, "An automatic formatting program for Pascal," *SIGPLAN Notices*, vol. 12, pp. 82-84, July 1977.
- [11] T.C.S. Kennedy, "The design of interactive procedures for man-machine communication," *Int. J. Man-Mach. Studies*, vol. 5, pp. 309-334, 1974.
- [12] J. S. Kidd and H. P. Van Cott, Eds., "System and human engineering analyses," in *Human Engineering Guide to Equipment Design*. Washington, DC: Gov. Printing Office, document D4.10:EN3, 1972, ch. 1.
- [13] H. Ledgard, A. Singer, and J. Whiteside, *Directions in Human Factors for Interactive Systems (Lecture Notes in Computer Science)*, no. 103. New York: Springer-Verlag, 1981.
- [14] W. C. Mann, "Why things are so bad for the computer-naive user," in *Proc. Nat. Comput. Conf.*, 1975, pp. 785-787.
- [15] L. Miller and J. C. Thomas, Jr., "Behavioral issues in the use of interactive systems: Part I. General system issues," Thomas J. Watson Res. Center, Yorktown Heights, NY, 1977.
- [16] J. Palme, "Interactive software for humans," Res. Inst. Nat. Defense, Stockholm, Sweden, NTIS PB-245 553, July 1976.
- [17] H. M. Parsons, "The scope of human factors in computer-based data processing systems," *Human Factors*, vol. 12, no. 2, pp. 165-175, 1970.
- [18] A. Singer, "Formal methods and human factors in the design of interactive systems," Ph.D. dissertation, Dep. Comput. Inform. Sci., Univ. Massachusetts, Amherst, 1979.
- [19] D. Stemple, private communication, 1975.
- [20] T. D. Sterling, "Guidelines for humanizing computerized information systems: A report from Stanley House," *Commun. Ass. Comput. Mach.*, vol. 17, Nov. 1974.
- [21] W. Teitelman, "Interlisp reference manual," Palo Alto Res. Center, Xerox Corp., Palo Alto, CA, 1974.
- [22] E. L. Thorndike and R. T. Rock, Jr., "Learning without awareness of what is being learned or intent to learn it," *J. Experimental Psychol.*, vol. XVII, no. 1, 1934.
- [23] R. Turner, "Interaction data from CS/2," Digital Equipment Corp., Maynard, MA, 1974.
- [24] J. D. Vandenberg, "Improved operating procedures manuals," *Ergonomics*, vol. 10, no. 2, pp. 114-120, 1967.
- [25] G. Weinberg, *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.
- [26] C. Wiedmann, *Handbook of APL Programming*. New York: Petrocelli, 1974.
- [27] W. Epstein and M. Arlinsky, "The interaction of syntactic structures and learning instructions," *Psychol. Sci.*, no. 3, 1965.
- [28] H. Ledgard, J. A. Whiteside, A. Singer, and W. Seymour, "The natural language of interactive systems," *Commun. Ass. Comput. Mach.*, vol. 23, Oct. 1908.
- [29] G. A. Miller, "The magical number seven plus or minus two: Some limits on our capacity for processing information," *Psychol. Rev.*, no. 63, pp. 81-97, 1956.
- [30] R. Weist and J. Dolezal, "The effect of violating phrase structure rules and selectional restrictions on TEP patterns," *Psychol. Sci.*, no. 27, 1972.

38

## Global Comments

TERRY ROBERTS

I appreciate the value of an attempt to design a system from the point of view of the user rather than of the implementor. But I wonder how you are going to know how well you have done. There are lots of people out in the world designing systems, and many of these people are under the impression that they are doing reasonably well by their users. How do you compare your results with theirs? A seat-of-pants feel for the clarity of the design, which is what I am getting from the annotations, does not prove much. For instance, you seem to think that your decision not to allow abbreviation of three dangerous keywords was the wrong one, but how can you really know? (see note 17).

My (limited) experience with automatic help systems has been frustrating because of an inability to communicate to the system what it is I want help with. These problems come in two major flavors: one is the time when there is something I know I should be able to do, but I do not know what the system calls it and so cannot ask about it. In your system, what would happen to a poor user who had forgotten the keyword "MAKE", for instance? The other problem I often encounter is that a system which takes its cue from my context is likely to be led astray because my confusion often arises from my being in the wrong mode and not knowing how to change context.

The "User's Guide" was written as a reference manual for people who already understand and have experience with text editing and other computerish operations. This is clear from the fact that, at least in the major part of the Guide, you merely list all of the relevant pieces of syntax and your decisions about the semantics, without putting it all within a framework of when each of these facts is actually relevant. The naive users that you start the manual talking to, on the other hand, would profit far more by being taken step-by-step through a realistic session; it means a whole lot more if they are shown what is actually being displayed, what sort of commands to give in what situations, what kind of help to expect from the Assistant, and so forth. Since (I hope) you designed this system by simulating actual user experiences, such a manual should not be too much of an extension of what you already have. And this frame of mind would make impossible such travesties as note 31: "It is not obvious from the User's Guide, but..."

A paper like this could benefit from discussing your design process, beyond the head-scratching and literature-reading that you allude to occasionally.

## Comments from a Letter Written on July 19, 1978

BEN SHINEIDERMAN

I enjoyed reading *The Annotated Assistant* and feel it is one of the better results of introspective/protocol research in the area of interactive systems design. Your annotations demon-

strate a sensitive awareness to the problems and you dramatically produce effective solutions. You give me one more piece of evidence that human factors engineering does make a difference and that it is in fact very difficult.

I appreciated your frank comments about your struggles. The user should not see just a first attempt at a language design but a carefully worked out product where only the tip of the iceberg of work shows. The trick is to make complex processes appear easy and that takes a great deal of effort. ("There is one art, no more, no less: to do all things with artlessness"—Piet Hein.)

I liked most of the command structure, the attention to lucid presentation (good documentation creates the impression of quality software and builds user confidence), the notion of positive reinforcement usage (but you should have documentation on the error and information messages as part of the manual) and the GRIPE command (all complex systems are processes not products and the mechanism for evolutionary refinement should be part of the system).

Having said all these laudatory things, I still found enough to complain about. The first and most serious complaint is in your explicit decision to create an anthropomorphized Assistant. The illusion of a friendly helpful partner is only appreciated by naive beginners who feel intimidated and feel possibly reassured by the existence of a clever and human/human machine. Knowledgeable users frown on such gaming and want to be explicitly in control of what happens. They know that the machine may goof or misinterpret and they want to be in charge. I believe that the computer should act as a mechanical device whose actions are entirely predictable. This means that the user is more in control, must take greater responsibility for errors and must be a bit more careful and a bit more knowledgeable. I believe this to be an important issue and hope that I can convince you to change your position. CAI systems which started out with friendly greetings and human illusions have been replaced by more mechanical looking programs which behave like reliable machines. I would like to argue this point out with you in person sometime.

Finally, I suggest even more attention to building community and obtaining feedback. I assume monitoring probes will be built into the system to detect error patterns and facilitate experimental data collection. Why not have a NEWS command [or maybe - GRIPE (the inverse of GRIPE)] which gives users information about changes, additions, systems schedule, etc.

## The Annotated Assistant

JAN WALKER

In general, I had difficulty reviewing this paper. I find that my comments were constantly shifting in perspective. I could not decide if I was reviewing the *Assistant design*, the *User's Guide Design*, the *authors' explanations of both in the annotations*, or the *process by which the system design evolved*. I think I am not alone in this schizophrenia; I think it is inherent in the paper.

The following paragraphs summarize my major concerns with the paper. I have attached by detailed notes. 39

*Human Engineering:* What do the authors mean by human engineering? They are currently (and should stop) using the words as a synonym for their own intuitions.

I would argue that this is not a human-engineered system but rather a carefully and thoughtfully crafted design. If you build a go-cart without performing any studies of materials or designs and without knowing some mechanical engineering, did you "engineer" it or did you build it? If it works, are you an engineer or just lucky?

Human engineering requires both knowledge of human characteristics and study of the context in which the humans will be performing. The authors reveal their lack of knowledge of the former with their dated and only marginally appropriate references to psychological literature. Their project would have benefitted greatly if someone from their psychology department had participated.

The authors fail to satisfy the requirement for an analysis of the context for the system. The only statement of purpose appears on the first page of the Guide: "This Assistant can help you create, manipulate, and execute PASCAL (sic) programs." In fact, the system must have been designed for the university campus/course assignment environment. In that environment, the context (where it comes from; who it is for, what it is for, and what it can do) is usually expressed in the first lecture. Even that orienting context is missing here.

In addition, the authors neglect to specify what problems their system addresses. They should discuss program editing as a conceptual activity rather than as a sequence of physical activities. They should discuss the kinds of information and interactions required for effective program debugging. They should discuss what kind of work they expect a student to do during the actual computer session (as opposed to desk preparation). This kind of information is imperative in order to justify and to evaluate particular "human engineering" decisions.

### *Problems with the Guide*

The User's Guide itself requires some "human engineering." It is plagued by inadequate organization and poor design decisions. It is clearly amateur writing, on a par with similar efforts in many college computer centers. It is written by programmers for people who think like themselves.

The authors have stated that one of their goals is "friendly" user documentation. Unfortunately, they seem to have equated "friendly" with "familiar and colloquial."

In a proper documentation effort, the document clearly identifies the capabilities of the system, the problems it can be applied to, and the intended users. Again, it requires the same kind of global task description as the rest of human engineering.

The authors should analyze their own document design (as in note 39) and attempt to justify the design as a whole and particular design decisions.

### *Problems with the Design*

The authors have described what sounds like a nice system—clearly better than many and probably, worse than some.

However, their intent was not just to design another system but to human engineer a system. In this effort, I think they have not been successful. They failed to grapple with the first fundamental problem of human engineering, i.e., description of the user task in functional terms.

The authors should describe their design more analytically. They should test some of the testable alternative solutions either in informal experiments or by watching people using different systems.

#### *Problems with the Manuscript as a Publication*

The paper is interesting but frustrating. In one sense, there is too much to read for not enough information. The annotation method of describing the Guide is spotty. Some very low-level details receive disproportionate detail while very high-level decisions remain undiscussed. The annotation method might be more successful if the thing being annotated exhibited a tight, high quality design.

If the authors want interim publication of this work, they should write a paper describing the task, discussing the major design decisions, using sections from the Guide as examples in the paper. As it stands, the paper overwhelms the reader with bottom level detail without providing the necessary orienting perspective for understanding and appreciating the contribution to knowledge by the authors.

### Excerpts from a Review

TOM LOVE AND DAN J. HENDERSON

... A second general weakness revolves around the decision to build a contemporary editor for teleprinters rather than cathode ray tubes. From a human factors perspective a line-at-a-time editor will not match a screen-oriented editor which allows one to see the full text and make modifications in context. The concept of the line is not the unit that a user thinks about for either text or programs. For text we think about words, sentences, paragraphs, and pages. For programs, we think about blocks of code and modules. Try to debug an unfamiliar program with a window. Professional programmers who are known for their debugging ability never look at individual lines of code until they have built up a mental picture of the structure of the program under consideration. Professional chess players, when asked to memorize an unfamiliar chess board, follow a similar strategy (Hunt and Love, 1973; DeGroot, 1965). Editors need to support their users' desire to see a body of text at any one time and see the result of their changes.

A telephone conversation with Andrew Singer revealed that the teleprinter orientation was a conscious decision. In 1975-1976, when the design work on the Assistant was being done, the teleprinter was the dominant user device. There is some question whether it will retain its superiority through the 1980's.

Singer, Ledgard, and Hueras describe the typical view of most computer scientists of experimentations: "In a sense, whenever we build a system today we conduct a human factors experiment." This statement from their introduction reflects an incomplete understanding of the basic concept of an experiment. An experiment requires that known variables be manipulated in known ways and that the results of these manipulations can be measured and evaluated against some standard. Manipulation of variables without either control or measurement does not constitute an experiment and might at best be a case study. They might have said instead that whenever we build a system today, we have the potential for learning something new about human factors.

Not all is lost, however, because experimental studies of new systems are feasible even though they have rarely been done in the past. A powerful experimental paradigm available for such experiments is the time series experiment. An interactive computer system in a marvelous vehicle for conducting real world experiments with lots of subjects over large time periods with very accurate data recording mechanisms. However, to conduct such large-scale experiments on significant software systems, one needs to build in the data capturing procedures during the initial design and development process rather than try to graft it on later. Available data include use and misuse of system features, patterns of use over time, and learning time and procedures for uninitiated users of the system.

Since it is true that our state of knowledge about the human factors aspects of interactive systems is insufficient to answer the myriad of questions asked by system designers, building into the system a capability to monitor its use, and to change in response to user demands, are essential design criteria. The recording and analyses of such data should not be done haphazardly, however. The best of contemporary technology in the fields of statistics and human factors need to be applied after initial release of the system, as well as during design.

### IV. WHO ARE THE USERS?

Many system designers believe they know a better way to solve a certain problem than those people who have been solving that problem every day for the past few years. Occasionally the system designer really does know a better way. More commonly, the system designer gets to creatively "re-design" the solution to fit the problem once a lot of code has already been developed. Singer, Ledgard, and Hueras have chosen to bet on the rare event that they know a better way.

Singer, Ledgard, and Hueras want to build a "standard" system to solve an "unstated" class of problems encountered by an "unstated" user population. Even repeated references to those overused and misused buzzwords software engineering and human engineering, do not compensate for the failure to ask the following key questions.

1) What problems currently exist that this new system will help solve?

2) Who has the problem?

To take an extreme example, suppose the class of problems is the simulation of planned hardware architectures and the class of users is Chinese mathematicians. Surely the PASCAL

Assistant is not the most desirable system for this situation. Or to take a more realistic example: suppose the user is a financial analyst developing reports based upon daily data obtained from branches all over the country. Does the PASCAL Assistant provide the capability and functionality needed? Words like "standard" need to be used with utmost care!

A substantial new software project needs to have a built-in plan for regular technical reviews by all involved parties. By regular, we mean reviews held weekly, biweekly, or at most monthly, but certainly not quarterly or annually. Such a technical review process needs to begin during the initial definition stage of the project. At GE Information Services, we proceed from a requirements analysis phase to a functional specification phase. During functional specification we specify the actual user interface, including syntax, and review this specification with the user as it is being developed—usually biweekly. These biweekly reviews continue through to deployment of all new software. Please note that these external user reviews are not the only reviews conducted. Internal technical reviews are conducted even more frequently to insure that we build systems that will solve the real problems of the users and that these systems are as error free as we can make them.

Singer, Ledgard, and Hueras do not describe any such analyses of requirements nor any experience with or plan for conducting internal or external reviews prior to building the complete system. Their approach is riskier than most commercial organizations can afford! Development efforts such as this require a better definition of the user population and the class of problems expected to be solved, coupled with regular technical reviews by all interested parties.

#### V. WE KNOW WHAT'S GOOD FOR YOU

While attempting to reference a few selected psychological research findings, Singer, Ledgard, and Hueras got quickly diverted and began to describe their new, standard system. Note 39, for example, cites their beliefs but references none of the very large number of relevant research projects (e.g., the recent Query-by-Example-versus SEQUEL studies).

Without either a review process or the rigor of justifying design decisions based upon convincing research evidence, one is left wondering if the authors really know what's good for us standard users. For example, do they really know how to do standard prettyprinting of PASCAL programs? Have they chosen single character commands which are both highly mnemonic and natural language independent? Is the simple file structure described really adequate for professional software developers? Do I really want to edit and immediately execute erroneous commands, or do I want to preview those commands before executing them? Do I really not need veto power over changes before they are made? Am I really willing to wait for each job to execute before doing anything else? What if I want 20 000 lines printed; do I do it at my terminal at 300 baud and wait 9+ hours, or do I need some command to redirect such print files?

Utmost care must be taken when someone writes a paper saying that they have solved the problems of some class of users. It is more convincing if:

- 1) the solution exists and has been used for real problems;
- 2) the users of the solution are convinced that it is substantially better than existing solutions; and
- 3) quantitative data exist to support the warm feeling reported by users.

Few existing software systems pass criterion 3) above, but if you admit usage data as evidence, more will pass this test. Examples of home-grown systems which have passed these criteria are the PASCAL language and the UNIX operating system. Both were really developed in reaction to adverse experiences by the authors in which they felt an equally desirable result could be achieved more simply and more elegantly. One wonders if a MULTICS or ALGOL 68 experience is not required by designers of new tools in order to appreciate all of the real constraints that need to be met by such tools.

#### VIII. CONCLUSIONS

Overall, our impression of the Assistant is that it is a step in the right direction. The rapidly increasing power and speed, and just as rapidly decreasing cost of computer hardware is stimulating us to provide users with computer systems that do not require professional programmers to generate useful results. This means that man-machine interfaces must become more human engineered in the years to come.

"Soft" though it may be, the science of psychology can help to provide quantitative answers to the critical questions dealing with what kinds of systems people can really use. The next few years should see the rapid growth of the discipline of software psychology, and results from the research it spawns will be of enormous value in the development of major systems. Singer, Ledgard, and Hueras understand the priorities of the next decade, and they have undertaken the design of a system for the 1980's based on the tools of the 1970's. Organizations with more resources than a single three-person team will want to apply current techniques of requirements definition and technical review to interactive systems intended for large-scale use. While improvements upon the Assistant are both possible and necessary, the direction they have taken is laudable.

Singer, Ledgard, and Hueras have described a system which is good and better than many currently available systems. It is not as good as it could be and does not reflect what is possible in 1978. Our belief is that they have introduced a very desirable art form to the field of user manual documentation. We expect this art form to outlast any of the technical design decisions described in the paper itself.

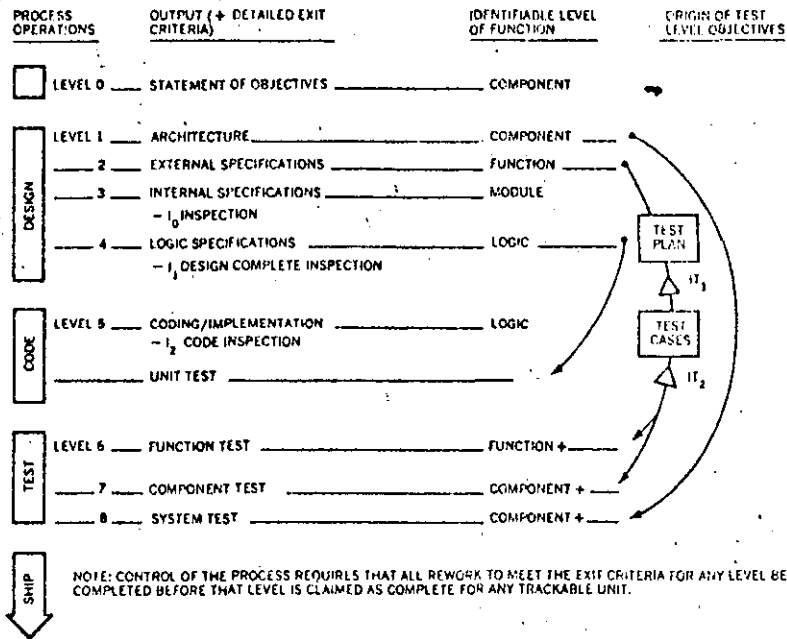
#### Author's Reply

A. SINGER, H. LEDGARD, AND J. F. HUERAS

We would like to express our appreciation to all the commentators for the considerable effort and thought that went into their remarks. Although we found ourselves at odds with



Figure 1 Programming process



inspected. A clear statement of the project rules and changes to these rules along with faithful adherence to the rules go a long way toward practicing the required project discipline.

A prerequisite of process management is a clearly defined series of operations in the process (Figure 1). The miniprocedure within each operation must also be clearly described for closer management. A clear statement of the criteria that must be satisfied to exit each operation is mandatory. This statement and accurate data collection, with the data clearly tied to trackable units of known size and collected from specific points in the process, are some essential constituents of the information required for process management.

In order to move the form of process management from qualitative to more quantitative, process terms must be more specific, data collected must be appropriate, and the limits of accuracy of the data must be known. The effect is to provide more precise information in the correct process context for decision making by the process manager.

In this paper, we first describe the programming process and places at which inspections are important. Then we discuss factors that affect productivity and the operations involved with inspections. Finally, we compare inspections and walk-throughs on process control.

## The process

a  
manageable  
process

3

A process may be described as a set of operations occurring in a definite sequence that operates on a given input and converts it to some desired output. A general statement of this kind is sufficient to convey the notion of the process. In a practical application, however, it is necessary to describe the input, output, internal processing, and processing times of a process in very specific terms if the process is to be executed and practical output is to be obtained.

In the programming development process, explicit requirement statements are necessary as input. The series of processing operations that act on this input must be placed in the correct sequence with one another, the output of each operation satisfying the input needs of the next operation. The output of the final operation is, of course, the explicitly required output in the form of a verified program. Thus, the objective of each processing operation is to receive a defined input and to produce a definite output that satisfies a specific set of exit criteria. (It goes without saying that each operation can be considered as a miniprocess itself.) A well-formed process can be thought of as a continuum of processing during which sequential sets of exit criteria are satisfied, the last set in the entire series requiring a well-defined end product. Such a process is not amorphous. It can be measured and controlled.

exit  
criteria

Unambiguous, explicit, and universally accepted exit criteria would be perfect as process control checkpoints. It is frequently argued that universally agreed upon checkpoints are impossible in programming because all projects are different, etc. However, *all* projects do reach the point at which there is a project checkpoint. As it stands, any trackable unit of code achieving a clean compilation can be said to have satisfied a universal exit criterion or checkpoint in the process. Other checkpoints can also be selected, albeit on more arguable premises, but once the premises are agreed upon, the checkpoints become visible in most, if not all, projects. For example, there is a point at which the design of a program is considered complete. This point may be described as the level of detail to which a unit of design is reduced so that one design statement will materialize in an estimated three to 10 source code instructions (or, if desired, five to 20, for that matter). Whichever particular ratio is selected across a project, it provides a checkpoint for the process control of that project. In this way, suitable checkpoints may be selected throughout the development process and used in process management. (For more specific exit criteria see Reference 1.)

The cost of reworking errors in programs becomes higher the later they are reworked in the process, so every attempt should be made to find and fix errors as early in the process as possible. This cost has led to the use of the inspections described later and to the description of exit criteria which include assuring that all errors known at the end of the inspection of the new "clean-compilation" code, for example, have been correctly fixed. So, rework of all known errors up to a particular point must be complete before the associated checkpoint can be claimed to be met for any piece of code.

4

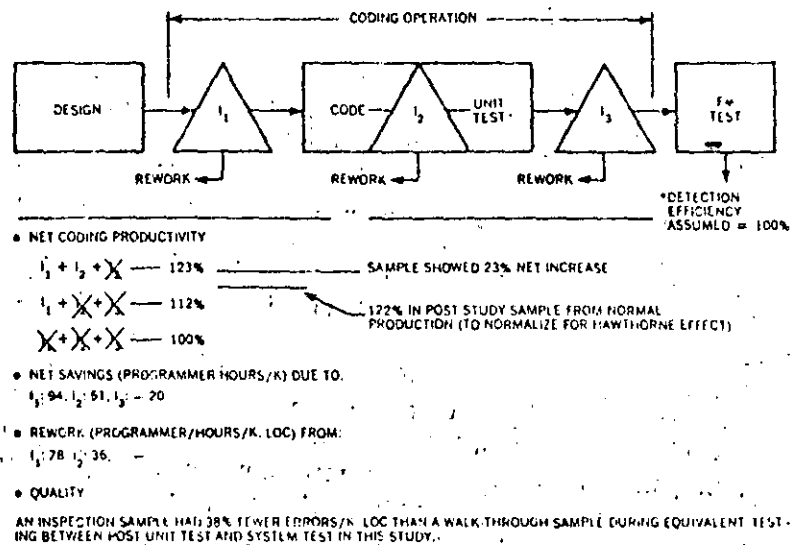
Where inspections are not used and errors are found during development or testing, the cost of rework as a fraction of overall development cost can be surprisingly high. For this reason, errors should be found and fixed as close to their place of origin as possible.

Production studies have validated the expected quality and productivity improvements and have provided estimates of standard productivity rates, percentage improvements due to inspections, and percentage improvements in error rates which are applicable in the context of large-scale operating system program production. (The data related to operating system development contained herein reflect results achieved by IBM in applying the subject processes and methods to representative samples. Since the results depend on many factors, they cannot be considered representative of every situation. They are furnished merely for the purpose of illustrating what has been achieved in sample testing.)

The purpose of the test plan inspection  $IT_1$ , shown in Figure 1, is to find voids in the functional variation coverage and other discrepancies in the test plan.  $IT_2$ , test case inspection of the test cases, which are based on the test plan, finds errors in the test cases. The total effects of  $IT_1$  and  $IT_2$  are to increase the integrity of testing and, hence, the quality of the completed product. And, because there are less errors in the test cases to be debugged during the testing phase, the overall project schedule is also improved.

A process of the kind depicted in Figure 1 installs all the intrinsic programming properties in the product as required in the statement of objectives (Level 0) by the time the coding operation (Level 5) has been completed—except for packaging and publications requirements. With these exceptions, all later work is of a verification nature. This verification of the product provides no contribution to the product during the essential development (Levels 1 to 5); it only adds error detection and elimination (frequently at one half of the development cost).  $I_0$ ,  $I_1$ , and  $I_2$  inspections were developed to measure and influence intrinsic

Figure 2 A study of coding productivity



quality (error content) in the early levels, where error rework can be most economically accomplished. Naturally, the beneficial effect on quality is also felt in later operations of the development process and at the end user's site.

An improvement in productivity is the most immediate effect of purging errors from the product by the  $I_0$ ,  $I_1$ , and  $I_2$  inspections. This purging allows rework of these errors very near their origin, early in the process. Rework done at these levels is 10 to 100 times less expensive than if it is done in the last-half of the process. Since rework detracts from productive effort, it reduces productivity in proportion to the time taken to accomplish the rework. It follows, then, that finding errors by inspection and reworking them earlier in the process reduces the overall rework time and increases productivity even within the early operations and even more over the total process. Since less errors ship with the product, the time taken for the user to install programs is less, and his productivity is also increased.

The quality of documentation that describes the program is of as much importance as the program itself for poor quality can mislead the user, causing him to make errors quite as important as errors in the program. For this reason, the quality of program documentation is verified by publications inspections ( $PI_0$ ,  $PI_1$ , and  $PI_2$ ). Through a reduction of user-encountered errors, these inspections also have the effect of improving user productivity by reducing his rework time.

Table 1 Error detection efficiency

Process Operations	Errors Found per K.NCSS	Percent of Total Errors Found
Design		
I <sub>1</sub> inspection	38*	82
Coding		
I <sub>2</sub> inspection		
Unit test		
Preparation for acceptance test	8	18
Acceptance test	0	
Actual usage (6 mo.)	0	
Total	46	100

\*51% were logic errors, most of which were missing rather than due to incorrect design.

8

In the development of applications, inspections also make a significant impact. For example, an application program of eight modules was written in COBOL by Aetna Corporate Data Processing department, Aetna Life and Casualty, Hartford, Connecticut, in June 1975.<sup>6</sup> Two programmers developed the program. The number of inspection participants ranged between three and five. The only change introduced in the development process was the I<sub>1</sub> and I<sub>2</sub> inspections. The program size was 4,439 Non-Commentary Source Statements.

Inspections in applications development

An automated estimating program, which is used to produce the normal program development time estimates for all the Corporate Data Processing department's projects, predicted that designing, coding, and unit testing this project would require 62 programmer days. In fact, the time actually taken was 46.5 programmer days including inspection meeting time. The resulting saving in programmer resources was 25 percent.

The inspections were obviously very thorough when judged by the inspection error detection efficiency of 82 percent and the later results during testing and usage as shown in Table 1.

The results achieved in Non-Commentary Source Statements per Elapsed Hour are shown in Table 2. These inspection rates are four to six times faster than for systems programming. If these rates are generally applicable, they would have the effect of making the inspection of applications programs much less expensive.

Table 2 Inspection rates in NCSS per hour

Operations	I <sub>1</sub>	I <sub>2</sub>
Preparation	898	709
Inspection	652	539

### Inspections

Inspections are a *formal, efficient, and economical* method of finding errors in design and code. All instructions are addressed

**Table 3. Inspection process and rate of progress**

Process operations	Rate of progress* (loc/hr)		Objectives of the operation
	Design I <sub>1</sub>	Code I <sub>2</sub>	
1. Overview	500	not necessary	Communication education
2. Preparation	100	125	Education
3. Inspection	130	150	Find errors
4. Rework	20 hrs/K.NCSS	16 hrs/K.NCSS	Rework and re-solve errors found by inspection
5. Follow-up	—	—	See that all errors, problems, and concerns have been resolved

\* These notes apply to systems programming and are conservative. Comparable rates for applications programming are much higher. Initial schedules may be started with these numbers and as project history that is keyed to unique environments evolves, the historical data may be used for future scheduling algorithms.

at least once in the conduct of inspections. Key aspects of inspections are exposed in the following text through describing the I<sub>1</sub> and I<sub>2</sub> inspection conduct and process. I<sub>0</sub>, IT<sub>1</sub>, IT<sub>2</sub>, PI<sub>0</sub>, PI<sub>1</sub>, and PI<sub>2</sub> inspections retain the same essential properties as the I<sub>1</sub> and I<sub>2</sub> inspections but differ in materials inspected, number of participants, and some other minor points.

the  
people  
involved

The inspection team is best served when its members play their particular roles, assuming the particular vantage point of those roles. These roles are described below:

1. *Moderator*—The key person in a successful inspection. He must be a competent programmer but need *not* be a technical expert on the program being inspected. To preserve objectivity and to increase the integrity of the inspection, it is usually advantageous to use a moderator from an unrelated project. The moderator must manage the inspection team and offer leadership. Hence, he must use personal sensitivity, tact, and drive in balanced measure. His use of the strengths of team members should produce a synergistic effect larger than their number; in other words, *he is the coach*. The duties of moderator also include scheduling suitable meeting places, reporting inspection results within one day, and follow-up on rework. *For best results the moderator should be specially trained.* (This training is brief but very advantageous.)
2. *Designer*—The programmer responsible for producing the program design.
3. *Coder/Implementor*—The programmer responsible for translating the design into code.
4. *Tester*—The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

If the coder of a piece of code also designed it, he will function in the designer role for the inspection process; a coder from some related or similar program will perform the role of the coder. If the same person designs, codes, and tests the product code, the coder role should be filled as described above, and another coder—preferably with testing experience—should fill the role of tester.

10

Four people constitute a good-sized inspection team, although circumstances may dictate otherwise. The team size should not be artificially increased over four, but if the subject code is involved in a number of interfaces, the programmers of code related to these interfaces may profitably be involved in inspection. Table 3 indicates the inspection process and rate of progress.

The total time to complete the inspection process from overview through follow-up for  $I_1$  or  $I_2$  inspections with four people involved takes about 90 to 100 people-hours for systems programming. Again, these figures may be considered conservative but they will serve as a starting point. Comparable figures for applications programming tend to be much lower, implying lower cost per K.NCSS.

scheduling  
inspections  
and rework

Because the error detection efficiency of most inspection teams tends to dwindle after two hours of inspection but then picks up after a period of different activity, it is advisable to schedule inspection sessions of no more than two hours at a time. Two two-hour sessions per day are acceptable.

The time to do inspections and resulting rework must be scheduled and managed with the same attention as other important project activities. (After all, as is noted later, for one case at least, it is possible to find approximately two thirds of the errors reported during an inspection.) If this is not done, the immediate work pressure has a tendency to push the inspections and/or rework into the background, postponing them or avoiding them altogether. The result of this short-term respite will obviously have a much more dramatic long-term negative effect since the finding and fixing of errors is delayed until later in the process (and after turnover to the user). Usually, the result of postponing early error detection is a lengthening of the overall schedule and increased product cost.

Scheduling inspection time for modified code may be based on the algorithms in Table 3 *and on judgment*.

Keeping the objective of each operation in the forefront of team activity is of paramount importance. Here is presented an outline of the  $I_1$  inspection process operations.

$I_1$   
inspection  
process

Figure 3 Summary of design inspections by error type

VP Individual Name	Inspection file			Errors	Error %
	Missing	Wrong	Extra		
CD CB Definition	16	2		18	3.5
CU CB Usage	18	17	1	36	6.9
FS FPFS	1			1	.2
IC Interconnect Calls	18	9		27	5.2
IR Interconnect Reqs	4	5	2	11	2.1
LO Logic	126	57	24	207	39.8
L3 Higher Lvl Docu	1		1	2	.4
MA Mod Attributes	1			1	.2
MD More Detail	24	6	2	32	6.2
MN Maintainability	8	5	3	16	3.1
OT Other	15	10	10	35	6.7
PD Pass Data Areas		1		1	.2
PE Performance	1	2	3	6	1.2
PR Prologue/Prose	44	38	7	89	17.1
RM Return Code/Msg	5	7	2	14	2.7
RU Register Usage	1	2		3	.6
ST Standards					
TB Test & Branch	12	7	2	21	4.0
	295	168	57	520	100.0
	57%	32%	11%		

Figure 4 Summary of code inspections by error type

VP Individual Name	Inspection file			Errors	Error %
	Missing	Wrong	Extra		
CC Code Comments	5	17	1	23	6.6
CU CB Usage	3	21	1	25	7.2
DE Design Error	31	32	14	77	22.1
FJ		8		8	2.3
IR Interconnect Calls	7	9	3	19	5.5
LO Logic	33	49	10	92	26.4
MN Maintainability	5	7	2	14	4.0
OT Other					
PE Performance	3	2	5	10	2.9
PR Prologue/Prose	25	24	3	52	14.9
PU PL/S or BAL Use	4	9	1	14	4.0
RU Register Usage	4	2		6	1.7
SU Storage Usage	1			1	.3
TB Test & Branch	2	5		7	2.0
	123	185	40	348	100.0

1. *Overview* (whole team)—The designer first describes the overall area being addressed and then the specific area he has designed in detail—logic, paths, dependencies, etc. Documentation of design is distributed to all inspection participants on conclusion of the overview. (For an I<sub>2</sub> inspection, no overview is necessary, but the participants should remain the same. Preparation, inspection, and follow-up proceed as for I<sub>1</sub>, but, of course, using code listings and design specifications



as inspection materials. Also, at  $I_2$  the moderator should flag for special scrutiny those areas that were reworked since  $I_1$  errors were found *and other design changes made.*

2. *Preparation* (individual) – Participants, using the design documentation, literally do their homework to try to understand the design, its intent and logic. (Sometimes flagrant errors are found during this operation, but in general, the number of errors found is not nearly as high as in the inspection operation.) To increase their error detection in the inspection, the inspection team should first study the ranked distributions of error types found by recent inspections. This study will prompt them to concentrate on the most fruitful areas. (See examples in Figures 3 and 4.) Checklists of clues on finding these errors should also be studied. (See partial examples of these lists in Figures 5 and 6 and complete examples for  $I_0$  in Reference 1 and for  $I_1$  and  $I_2$  in Reference 7.)
3. *Inspection* (whole team) – A “reader” chosen by the moderator (usually the coder) describes how he will implement the design. He is expected to paraphrase the design as expressed by the designer. Every piece of logic is covered at least once, and every branch is taken at least once. All higher-level documentation, high-level design specifications, logic specifications, etc., and macro and control block listings at  $I_2$  must be available and present during the inspection.

Now that the design is understood, *the objective is to find errors.* (Note that an error is defined as any condition that causes malfunction or that precludes the attainment of expected or previously specified results. Thus, deviations from specifications are clearly termed errors.) The finding of errors is actually done during the implementor/coder's discourse. Questions raised are pursued only to the point at which an error is recognized. It is noted by the moderator: its type is classified; severity (major or minor) is identified, and the inspection is continued. Often the solution of a problem is obvious. If so, it is noted, but no specific solution hunting is to take place during inspection. (The inspection is *not* intended to redesign, evaluate alternate design solutions, or to find solutions to errors; it is intended just to find errors!) A team is most effective if it operates with only one objective at a time.

Within one day of conclusion of the inspection, the moderator should produce a written report of the inspection and its findings to ensure that all issues raised in the inspection will be addressed in the rework and follow-up operations. Examples of these reports are given as Figures 7A, 7B, and 7C.

Figure 5 Examples of what to examine when looking for errors at I<sub>1</sub>

I<sub>1</sub> Logic

13

*Missing*

1. Are All Constants Defined?
2. Are All Unique Values Explicitly Tested on Input Parameters?
3. Are Values Stored after They Are Calculated?
4. Are All Defaults Checked Explicitly Tested on Input Parameters?
5. If Character Strings Are Created Are They Complete. Are All Delimiters Shown?
6. If a Keyword Has Many Unique Values, Are They All Checked?
7. If a Queue Is Being Manipulated, Can the Execution Be Interrupted: If So, Is Queue Protected by a Locking Structure: Can Queue Be Destroyed Over an Interrupt?
8. Are Registers Being Restored on Exits?
9. In Queuing/Dequeuing Should Any Value Be Decremented/Incremented?
10. Are All Keywords Tested in Macro?
11. Are All Keyword Related Parameters Tested in Service Routine?
12. Are Queues Being Held in Isolation So That Subsequent Interrupting Requestors Are Receiving Spurious Returns Regarding the Held Queue?
13. Should any Registers Be Saved on Entry?
14. Are All Increment Counts Properly Initialized (0 or 1)?

*Wrong*

1. Are Absolutes Shown Where There Should Be Symbolics?
2. On Comparison of Two Bytes, Should All Bits Be Compared?
3. On Built Data Strings, Should They Be Character or Hex?
4. Are Internal Variables Unique or Confusing If Concatenated?

*Extra*

1. Are All Blocks Shown in Design Necessary or Are They Extraneous?

4. *Rework*—All errors or problems noted in the inspection report are resolved by the designer or coder/implementor.

5. *Follow-Up*—It is imperative that every issue, concern, and error be entirely resolved at this level, or errors that result can be 10 to 100 times more expensive to fix, if found later in the process (programmer time only, machine time not included). It is the responsibility of the moderator to see that all issues, problems, and concerns discovered in the inspection operation have been resolved by the designer in the case of I<sub>1</sub>, or the coder/implementor for I<sub>2</sub> inspections. If more than five percent of the material has been reworked, the team should reconvene and carry out a 100-percent reinspection. Where less than five percent of the material has been reworked, the moderator at his discretion may verify the quality of the rework himself or reconvene the team to reinspect either the complete work or just the rework.

commencing  
inspections

In Operation 3 above, it is one thing to direct people to find errors in design or code. It is quite another problem for them to find errors. Numerous experiences have shown that people have to be taught or prompted to find errors effectively. Therefore, it

Figure 6 Examples of what to examine when looking for errors at I<sub>2</sub>

INSPECTION SPECIFICATION

I<sub>2</sub> Test Branch

- Is Correct Condition Tested (IF X = ON vs. IF X = OFF)?
- Is (Are) Correct Variable(s) Used for Test (IF X = ON vs. IF Y = ON)?
- Are Null THENS/ELSEs Included as Appropriate?
- Is Each Branch Target Correct?
- Is the Most Frequently Exercised Test leg the THEN Clause?

14

I<sub>2</sub> Interconnection (or Linkage) Calls

- For Each Interconnection Call to Either a Macro, SVC or Another Module:
  - Are All Required Parameters Passed Set Correctly?
  - If Register Parameters Are Used, Is the Correct Register Number Specified?
  - If Interconnection Is a Macro,
    - Does the Inline Expansion Contain All Required Code?
    - No Register or Storage Conflicts between Macro and Calling Module?
    - If the Interconnection Returns, Do All Returned Parameters Get Processed Correctly?

is prudent to condition them to seek the high-occurrence, high-cost error types (see example in Figures 3 and 4), and then describe the clues that usually betray the presence of each error type (see examples in Figures 5 and 6).

One approach to getting started may be to make a preliminary inspection of a design or code that is felt to be representative of the program to be inspected. Obtain a suitable quantity of errors, and analyze them by type and origin, cause, and salient indicative clues. With this information, an inspection specification may be constructed. This specification can be amended and improved in light of new experience and serve as an on-going directive to focus the attention and conduct of inspection teams. The objective of an inspection specification is to help maximize and make more consistent the error detection efficiency of inspections where

Error detection efficiency

$$= \frac{\text{Errors found by an inspection}}{\text{Total errors in the product before inspection}} \times 100$$

The reporting forms and form completion instructions shown in the Appendix may be used for I<sub>1</sub> and I<sub>2</sub> inspections. Although these forms were constructed for use in systems programming development, they may be used for applications programming development with minor modification to suit particular environments.

reporting  
inspection  
results

The moderator will make hand-written notes recording errors found during inspection meetings. He will categorize the errors

Figure 7A Error list

1. PR/M/MIN Line 3: the statement of the prologue in the REMARKS section needs expansion.
2. DA/W/MAJ Line 123: ERR-RECORD-TYPE is out of sequence. 15
3. PU/W/MAJ Line 147: the wrong bytes of an 8-byte field (current-data) are moved into the 2-byte field (this year).
4. LO/W/MAJ Line 169: while counting the number of leading spaces in NAME, the wrong variable (I) is used to calculate "J".
5. LO/W/MAJ Line 172: NAME-CHECK is PERFORMED one time too few.
6. PU/E/MIN Line 175: In NAME-CHECK, the check for SPACE is redundant.
7. DE/W/MIN Line 175: the design should allow for the occurrence of a period in a last name.

Figure 7B Example of module detail report

DATE \_\_\_\_\_

CODE INSPECTION REPORT  
MODULE DETAIL

MOD/MAC: CHECKER \_\_\_\_\_ SUBCOMPONENT/APPLICATION \_\_\_\_\_

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LD LOGIC _____		9			1	
TB: TEST AND BRANCH _____						
EL: EXTERNAL LINKAGES _____						
RU: REGISTER USAGE _____						
SU: STORAGE USAGE _____						
DA: DATA AREA USAGE _____		2				
PU: PROGRAM LANGUAGE _____		2				1
PE: PERFORMANCE _____					1	
MN: MAINTAINABILITY _____					1	
DE: DESIGN ERROR _____					1	
PR: PROLOGUE _____				1		
CC: CODE COMMENTS _____						
OT: OTHER _____						
TOTAL:		13			5	

REINSPECTION REQUIRED?  Y \_\_\_\_\_

\*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION IS RATED M - MISSING, W - WRONG, E - EXTRA.  
NOTE: FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER: (M) WHERE (M) IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND (E) IS THE NUMBER OF PROBLEMS IN THE BASE.

and then transcribe counts of the errors, by type, to the module detail form. By maintaining cumulative totals of the counts by error type, and dividing by the number of projected executable source lines of code inspected to date, he will be able to establish installation averages within a short time.

Figures 7A, 7B, and 7C are an example of a set of code inspection reports. Figure 7A is a partial list of errors found in code inspection. Notice that errors are described in detail and are classified by error type, whether due to something being missing,

Figure 7C Example of code inspection summary report

**CODE INSPECTION REPORT  
SUMMARY**

Date: 11/20/-

To: Design Manager: KRAUSS Development Manager: GIOIT

Subject: Inspection Report for: CHECKER Inspection date: 11/19/-

System/Application: \_\_\_\_\_ Release: \_\_\_\_\_ Build: \_\_\_\_\_

Component: \_\_\_\_\_ Subcomponent(s): \_\_\_\_\_

Mod/Mac Name	New or Mod	Full or Part Insp	Programmer	Tester	ELOC									Inspection				Sub-component			
					Added			Modified			Deleted			Prep	Insp	Rework	Follow-up				
					A	M	D	A	M	D	A	M	D						People-hours (x.x)		
	N		MCGINLEY	HALE	348			400						50			9.0	8.8	8.0	1.5	
<b>Totals</b>																					

Reinspection required? YES Length of inspection (clock hours and tenths) 2.2

Reinspection by (date) 11/25/- Additional modules/macros? NO

DCR #'s written C:2

Problem summary: Major 13 Minor 5 Total 18

Errors in changed code: Major \_\_\_\_\_ Minor \_\_\_\_\_ Errors in base code: Major \_\_\_\_\_ Minor \_\_\_\_\_

LARSON \_\_\_\_\_ MCGINLEY \_\_\_\_\_ HALE \_\_\_\_\_

Initial Desr: \_\_\_\_\_ Detailed Or: \_\_\_\_\_ Programmer: \_\_\_\_\_ Team Leader: \_\_\_\_\_ Other: \_\_\_\_\_ Moderator's Signature: \_\_\_\_\_

16

wrong, or extra as the cause, and according to major or minor severity. Figure 7B is a module level summary of the errors contained in the entire error list represented by Figure 7A. The code inspection summary report in Figure 7C is a summary of inspection results obtained on all modules inspected in a particular inspection session or in a subcomponent or application.

Inspections have been successfully applied to designs that are specified in English prose, flowcharts, HIPO, (Hierarchy plus Input-Process-Output) and PIDGEON (an English prose-like meta language).

inspections  
and  
languages

The first code inspections were conducted on PL/S and Assembler. Now, prompting checklists for inspections of Assembler, COBOL, FORTRAN, and PL/I code are available.<sup>7</sup>

One of the most significant benefits of inspections is the detailed feedback of results on a relatively real-time basis. The programmer finds out what error types he is most prone to make and their quantity and how to find them. This feedback takes place within a few days of writing the program. Because he gets early indications from the first few units of his work inspected, he is able to show improvement, and usually does, on later work even during the same project. In this way, feedback of results from inspections must be counted for the programmer's use and benefit: *they should not under any circumstances be used for programmer performance appraisal.*

personnel  
considerations

Skeptics may argue that once inspection results are obtained, they will or even must count in performance appraisals, or at

Figure 8 Example of most error prone modules based on  $I_1$  and  $I_2$

Module name	Number of errors	Lines of code	Error density, Errors/K.Loc
Echo	4	128	31
Zulu	10	323	31
Foxtrot	3	71	28
Alpha	7	264	27
Lima	2	106	19
Delta	3	195	15
			27 - Average Error Rate
	67		

**17**

least cause strong bias in the appraisal process. The author can offer in response that inspections have been conducted over the past three years involving diverse projects and locations, hundreds of experienced programmers and tens of managers, and so far he has found no case in which inspection results have been used negatively against programmers. Evidently no manager has tried to "kill the goose that lays the golden eggs."

A preinspection opinion of some programmers is that they do not see the value of inspections because they have managed very well up to now, or because their projects are too small or somehow different. This opinion usually changes after a few inspections to a position of acceptance. The quality of acceptance is related to the success of the inspections they have experienced, the *conduct of the trained moderator*, and the *attitude demonstrated by management*. The acceptance of inspections by programmers and managers as a beneficial step in making programs is well-established amongst those who have tried them.

#### Process control using inspection and testing results

Obviously, the range of analysis possible using inspection results is enormous. Therefore, only a few aspects will be treated here, and they are elementary expositions.

most  
error-prone  
modules

A listing of either  $I_1$ ,  $I_2$ , or combined  $I_1 + I_2$  data as in Figure 8 immediately highlights which modules contained the highest error density on inspection. If the error detection efficiency of each of the inspections was fairly constant, the ranking of error-prone modules holds. Thus if the error detection efficiency of inspection is 50 percent, and the inspection found 10 errors in a

Figure 9 Example of distribution of error types

	Number of errors	%	Normal/usual distribution, %
Logic	23	35	44
Interconnection/Linkage (Internal)	21	31 ?	18
Control Blocks	6	9	13
—	.	8	10
—	.	7	7
—	.	6	6
—	.	4	2
		100%	100%

18

module, then it can be estimated that there are 10 errors remaining in the module. This information can prompt many actions to control the process. For instance, in Figure 8, it may be decided to reinspect module "Echo" or to redesign and recode it entirely. Or, less drastically, it may be decided to test it "harder" than other modules and look especially for errors of the type found in the inspections.

If a ranked distribution of error types is obtained for a group of "error-prone modules" (Figure 9), which were produced from the same Process A, for example, it is a short step to comparing this distribution with a "Normal/Usual Percentage Distribution." Large disparities between the sample and "standard" will lead to questions on why Process A, say, yields nearly twice as many internal interconnection errors as the "standard" process. If this analysis is done promptly on the first five percent of production, it may be possible to remedy the problem (if it is a problem) on the remaining 95 percent of modules for a particular shipment. Provision can be made to test the first five percent of the modules to remove the unusually high incidence of internal interconnection problems.

distribution of error types

Analysis of the testing results, commencing as soon as testing errors are evident, is a vital step in controlling the process since future testing can be guided by early results.

inspecting error-prone code

Where testing reveals excessively error-prone code, it may be more economical and saving of schedule to select the most error-prone code and inspect it before continuing testing. (The business case will likely differ from project to project and case to case, but in many instances inspection will be indicated). The selection of the most error-prone code may be made with two considerations uppermost:

Table 4. Inspection and walk-through processes and objectives

<i>Inspection</i>		<i>Walk-through</i>	
<i>Process Operations</i>	<i>Objectives</i>	<i>Process Operations</i>	<i>Objectives</i>
1. Overview	Education (Group)	-	-
2. Preparation	Education (Individual)	1. Preparation	Education (Individual)
3. Inspection	Find errors! (Group)	2. Walk-through	Education (Group) Discuss design alternatives Find errors
4. Rework	Fix problems	-	
5. Follow-up	Ensure all fixes correctly installed	-	

Note the separation of objectives in the inspection process.

Table 5 Comparison of key properties of inspections and walk-throughs

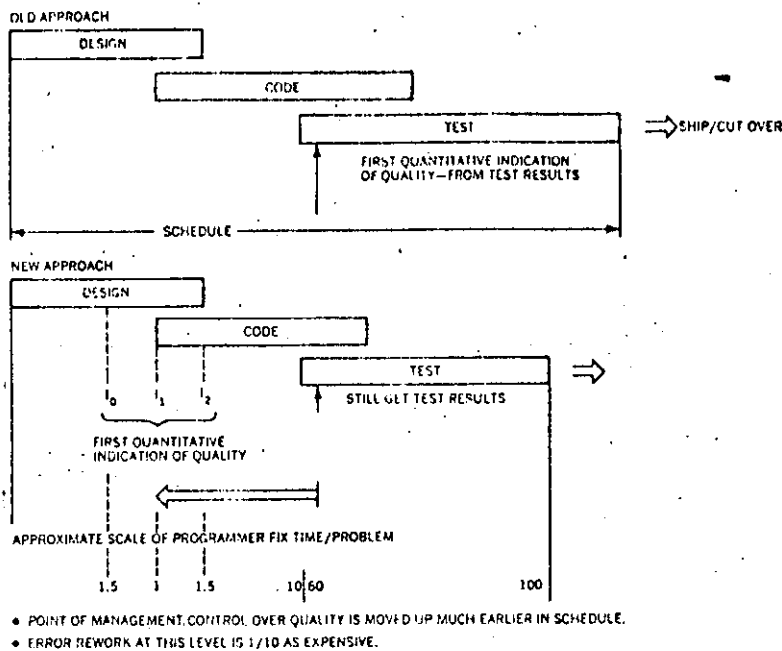
<i>Properties</i>	<i>Inspection</i>	<i>Walk-Through</i>
1. Formal moderator training	Yes	No
2. Definite participant roles	Yes	No
3. Who "drives" the inspection or walk-through	Moderator	Owner of material (Designer or coder)
4. Use "How To Find Errors" checklists	Yes	No
5. Use distribution of error types to look for	Yes	No
6. Follow-up to reduce bad fixes	Yes	No
7. Less future errors because of detailed error feedback to individual programmer	Yes	Incidental
8. Improve inspection efficiency from analysis of results	Yes	No
9. Analysis of data → process problems → improvements	Yes	No

1. Which modules head a ranked list when the modules are rated by test errors per K.NCSS?
2. In the parts of the program in which test coverage is low, which modules or parts of modules are most suspect based on  $(I_1 + I_2)$  errors per K.NCSS and programmer judgment?

From a condensed table of ranked "most error-prone" modules, a selection of modules to be inspected (or reinspected) may be made. Knowledge of the error types already found in these modules will better prepare an inspection team.



Figure 11 Effect of inspection on process management



made early in the process (during the first half of the project instead of the latter half of the schedule, when recovery may be impossible without adjustments in schedule and cost). Since individually trackable modules of reasonably well-known size can be counted as they pass through each of these checkpoints, the percentage completion of the project against schedule can be continuously and easily tracked.

The overview, preparation, and inspection sequence of the operations of the inspection process give the inspection participants a high degree of product knowledge in a very short time. This important side benefit results in the participants being able to handle later development and testing with more certainty and less false starts. Naturally, this also contributes to productivity improvement.

effect on  
product  
knowledge

An interesting sidelight is that because designers are asked at pre- $I_1$  inspection time for estimates of the number of lines of code (NCSS) that their designs will create, and they are present to count for themselves the actual lines of code at the  $I_2$  inspection, the accuracy of design estimates has shown substantial improvement.

For this reason, an inspection is frequently a required event where responsibility for design or code is being transferred from

one programmer to another. The complete inspection team is convened for such an inspection. (One-on-one reviews such as desk debugging are certainly worthwhile but do not approach the effectiveness of formal inspection.) Usually the side benefit of finding errors more than justifies the transfer inspection.

23

inspecting  
modified  
code

Code that is changed in, or inserted in, an existing module either in replacement of deleted code or simply inserted in the module is considered modified code. By this definition, a very large part of programming effort is devoted to modifying code. (The addition of entirely new modules to a system count as new, not modified, code.)

Some observations of errors per K.NCSS of modified code show its error rate to be considerably higher than is found in new code; (i.e., if 10.NCSS are replaced in a 100.NCSS module and errors against the 10.NCSS are counted, the error rate is described as number of errors per 10.NCSS, not number of errors per 100.NCSS). Obviously, if the number of errors in modified code are used to derive an error rate per K.NCSS for the whole module that was modified, this rate would be largely dependent upon the percentage of the module that is modified; this would provide a meaningless ratio. A useful measure is the number of errors per K.NCSS (modified) in which the higher error rates have been observed.

Since most modifications are small (e.g., 1 to 25 instructions), they are often erroneously regarded as trivially simple and are handled accordingly; the error rate goes up, and control is lost. In the author's experience, *all* modifications are well worth inspecting from an economic and a quality standpoint. A convenient method of handling changes is to group them to a module or set of modules and convene the inspection team to inspect as many changes as possible. *But all changes must be inspected!*

Inspections of modifications can range from inspecting the modified instructions and the surrounding instructions connecting it with its host module, to an inspection of the entire module. The choice of extent of inspection coverage is dependent upon the percentage of modification, pervasiveness of the modification, etc.

bad  
fixes

A very serious problem is the inclusion in the product of bad fixes. Human tendency is to consider the "fix," or correction, to a problem to be error-free itself. Unfortunately, this is all too frequently untrue in the case of fixes to errors found by inspections and by testing. The inspection process clearly has an operation called Follow-Up to try and minimize the bad-fix problem, but the fix process of testing errors very rarely requires scrutiny of fix quality before the fix is inserted. Then, if the fix is bad, the whole elaborate process of going from source fix to link edit, to

test the fix, to regression test must be repeated at needlessly high cost. The number of bad fixes can be economically reduced by some simple inspection after clean compilation of the fix.

## Summary

24

We can summarize the discussion of design and code inspections and process control in developing programs as follows:

1. Describe the program development process in terms of operations, and define exit criteria which must be satisfied for completion of each operation.
2. Separate the objectives of the inspection process operations to keep the inspection team focused on one objective at a time:

<i>Operation</i>	<i>Objective</i>
Overview	Communications/education
Preparation	Education
Inspection	Find errors
Rework	Fix errors
Follow-up	Ensure all fixes are applied correctly

3. Classify errors by type, and rank frequency of occurrence of types. Identify *which types* to spend most time looking for in the inspection.
4. Describe *how* to look for presence of error types.
5. Analyze inspection results and use for constant process improvement (until process averages are reached and then use for process control).

Some applications of inspections include function level inspections  $I_0$ , design-complete inspections  $I_1$ , code inspections  $I_2$ , test plan inspections  $IT_1$ , test case inspections  $IT_2$ , interconnections inspections  $IF$ , inspection of fixes/changes, inspection of publications, etc., and post testing inspection. Inspections can be applied to the development of system control programs, applications programs, and microcode in hardware.

We can conclude from experience that inspections increase productivity and improve final program quality. Furthermore, improvements in process control and project management are enabled by inspections.

## ACKNOWLEDGMENTS

The author acknowledges, with thanks, the work of Mr. O. R. Kohli and Mr. R. A. Radice, who made considerable contributions in the development of inspection techniques applied to program design and code, and Mr. R. R. Larson, who adapted inspections to program testing.

Figure 12 Design inspection module detail form

DATE: \_\_\_\_\_

DETAILED DESIGN INSPECTION REPORT  
MODULE DETAIL

MOD/MAC: \_\_\_\_\_ SUBCOMPONENT/APPLICATION: \_\_\_\_\_ 25

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LO: LOGIC						
TB: TEST AND BRANCH						
DA: DATA AREA USAGE						
RM: RETURN CODES MESSAGES						
RU: REGISTER USAGE						
MA: MODULE ATTRIBUTES						
EL: EXTERNAL LINKAGES						
MD: MORE DETAIL						
ST: STANDARDS						
PR: PROLOGUE OR PROSE						
HL: HIGHER LEVEL DESIGN DOC.						
US: USER SPEC.						
MN: MAINTAINABILITY						
PE: PERFORMANCE						
OT: OTHER						
TOTAL:						

REINSPECTION REQUIRED? \_\_\_\_\_

\*A PROBLEM WHICH AFFECTS THE PROGRAM TO MALFUNCTION IS A BIG M. A MISSING WORD, WHICH IS EXTRA, IS A MODIFIED WORD, WHICH IS IN THE CHARACTER POSITION, IS A SMALL M. THE PROBLEMS IN THE PAGE SHOULD BE SHOWN IN THIS MANNER: 23, WHERE 23 IS THE NUMBER OF PROBLEMS IN THE CHANGED POSITION AND 2 IS THE NUMBER OF PROBLEMS IN THE PAGE.

CITED REFERENCES AND FOOTNOTES

1. O. R. Kohli, *High-Level Design Inspection Specification*. Technical Report TR 21.601, IBM Corporation, Kingston, New York (July 21, 1975).
2. It should be noted that the exit criteria for  $I_1$  (design complete where one design statement is estimated to represent 3 to 10 code instructions) and  $I_2$  (first clean code compilations) are checkpoints in the development process through which every programming project must pass.
3. The Hawthorne Effect is a psychological phenomenon usually experienced in human-involved productivity studies. The effect is manifested by participants producing above normal because they know they are being studied.
4. NCSS (Non-Commentary Source Statements), also referred to as "Lines of Code," are the sum of executable code instructions and declaratives. Instructions that invoke macros are counted once only. Expanded macroinstructions are also counted only once. Comments are not included.
5. Basically in a walk-through, program design or code is reviewed by a group of people gathered together at a structured meeting in which errors/issues pertaining to the material and proposed by the participants may be discussed in an effort to find errors. The group may consist of various participants but always includes the originator of the material being reviewed who usually plans the meeting and is responsible for correcting the errors. How it differs from an inspection is pointed out in Tables 2 and 3.
6. *Marketing Newsletter*, Cross Application Systems Marketing, "Program inspections at Actna," MS-76-006, S2, IBM Corporation, Data Processing Division, White Plains, New York (March 29, 1976).

4. REWORK ELOC: The estimated noncommentary source lines of code in rework as a result of the inspection.
5. PREP: The number of people hours (in tenths of hours) spent in preparing for the inspection meeting.

30

Reprint Order No. G321-5033.

# THE UNIT DEVELOPMENT FOLDER (UDF): AN EFFECTIVE MANAGEMENT TOOL FOR SOFTWARE DEVELOPMENT

*Prepared by  
Frank S. Ingrassia*

*October 1976*

**TRW**

DEFENSE AND SPACE SYSTEMS GROUP

SYSTEMS ENGINEERING AND INTEGRATION DIVISION

ONE SPACE PARK, REDONDO BEACH, CALIFORNIA 90278

© TRW, INC., 1976  
All Rights Reserved

**ABSTRACT**

This paper describes the content and application of the Unit Development Folder, a structured mechanism for organizing and collecting software development products (requirements, design, code, test plans/data) as they become available. Properly applied, the Unit Development Folder is an important part of an orderly development environment in which unit-level schedules and responsibilities are clearly delineated and their step-by-step accomplishment made visible to management. Unit Development Folders have been used on a number of projects at TRW and have been shown to reduce many of the problems associated with the development of software.

One of the main side effects resulting from the invention of computers has been the creation of a new class of frustrated and harried managers responsible for software development. The frustration is a result of missed schedules, cost overruns, inadequate implementation and design, high operational error rates and poor maintainability, which have historically characterized software development. In the early days of computer programming, these problems were often excused by the novelty of this unique endeavor and obscured by the language and experience gap that frequently existed between developers and managers. Today's maturity and the succession of computer-wise people to management positions does not appear to have reduced the frustration level in the industry. We are still making the same mistakes and getting into the same predicaments. The science of managing software development is still in its infancy and the lack of a good clear set of principles is apparent.

The problems associated with developing software are too numerous and too complex for anyone to pretend to have solved them, and this paper makes no such pretensions. The discussion that follows describes a simple but effective management tool which, when properly used, can reduce the chaos and alleviate many of the problems common to software development. The tool described in this paper is called the Unit Development Folder (UDF) and is being used at TRW in software development and management.

What is a UDF? Simply stated, it is a specific form of development notebook which has proven useful and effective in collecting and organizing software products as they are produced. In essence, however, it is much more; it is a means of imposing a management philosophy and a development methodology on an activity that is often chaotic. In physical appearance, a UDF is merely a three-ring binder containing a cover sheet and is organized into several predefined sections which are common to each UDF. The ultimate objectives that the content and format of the UDF must satisfy are to:

- (1) Provide an orderly and consistent approach in the development of each of the units of a program or project
- (2) Provide a uniform and visible collection point for all unit documentation and code
- (3) Aid individual discipline in the establishment and attainment of scheduled unit-level milestones
- (4) Provide low-level management visibility and control over the development process

Figure 1 illustrates the role of the UDF in the total software development process.

If one follows a fairly standard design approach, the completion of the preliminary design activity marks the point at which UDFs are created and initiated for all units comprising the total product to be designed and coded. Therefore, the first question to be answered is, "What is a unit?" It was found that, for the purpose of implementing a practical and effective software development methodology to meet the management objectives stated earlier, a unique element of software architecture needed to be defined. This basic functional element is designated a "unit" of software and is defined independently of the language or type of application. Experience has indicated that it is unwise to attempt a simple-minded definition which will be useful and effective in all situations. What can be done is to bound the problem by means of some general considerations and delegate the specific implementation to management judgment for each particular application.



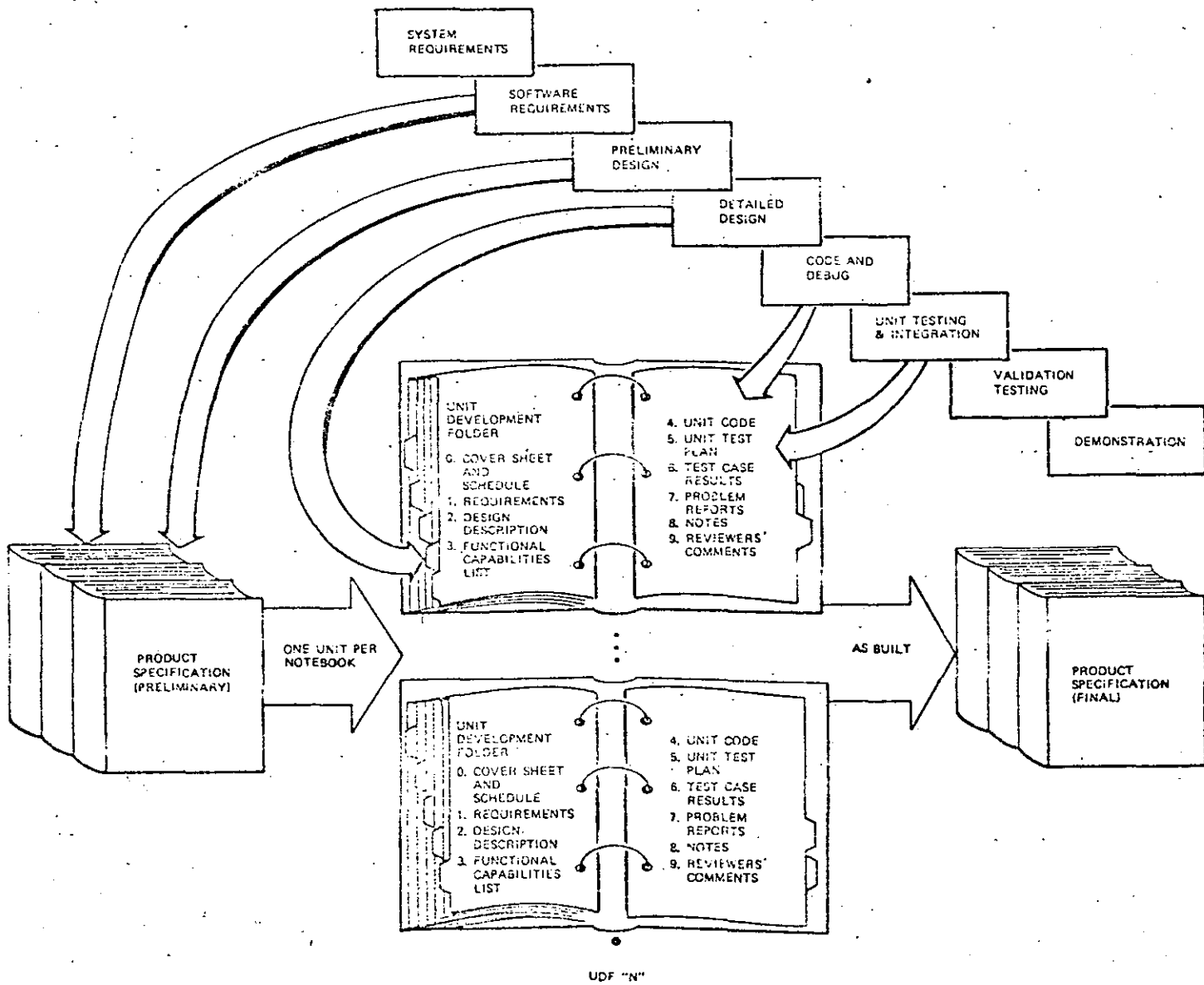


Figure 1. The UDF in the Development Process

At the lower end of the scale a "unit" can be defined to be a single routine or subroutine. At the upper end of the scale a "unit" may contain several routines comprising a subprogram or module. However it is defined, a unit of software should possess the following characteristics:

35

- (1) It performs a specific defined function
- (2) It is amenable to development by one person within the assigned schedule
- (3) It is a level of software to which the satisfaction of requirements can be traced
- (4) It is amenable to thorough testing in a disciplined environment.

The key word in the concept is manageability – in design, development, testing and comprehension.

A natural question that may arise at this point is, "Why should a unit contain more than one routine?" The assumption for this proviso is that the design and development standards impose both size and functional modularity. Since functional modularity can be defined at various levels, the concept can become meaningless if it is not accompanied by a reasonable restriction of size. Consequently, the maximum size constraint on routines may sometimes result in multiple-routine units.

The organization and content of a UDF can be adapted to reflect local conditions or individual project requirements. The important considerations in the structuring of a UDF are:

- (1) The number of subdivisions is not so large as to be confusing or unmanageable
- (2) Each of the sections contributes to the management and visibility of the development process
- (3) The content and format of each section are adequately and unambiguously defined
- (4) The subdivisions are sufficiently flexible to be applicable to a variety of software types
- (5) The individual sections are chronologically ordered as nearly as possible.

The last item is very important since it is this aspect of the UDF that relates it to the development schedule and creates an auditable management instrument. An example of a typical cover sheet for a UDF is shown in Figure 2; the contents of each section will be briefly described in subsequent paragraphs.

The UDF is initiated when requirements are allocated to the unit level and at the onset of preliminary design. At this point it exists in the skeletal form of a binder with a cover sheet (indicating the unit name and responsible custodian) and a set of section separators. The first step in the process is for the responsible work area manager to integrate the development schedules and responsibilities for each of his UDF's into the overall schedule and milestones of the project. A due date is generated for the completion of each section and the responsibility for each section is assigned. The originators should participate in establishing their interim schedules within the constraints of the dictated end dates.

The organization and subdivisions of the UDF are such that the UDF can accommodate a variety of development plans and approaches; it can be used in a situation where one person has total responsibility, or in the extreme where specialists are assigned to the particular sections. However, in the one-man approach it is still desirable that certain sections, indicated in the following discussion, be assigned to other individuals to gain the benefits of unbiased reviews and assessments.

The development of the UDF is geared to proceed logically and sequentially, and each section should be as complete as possible before proceeding to the next section. This is not always possible, and software development is usually an iterative rather than a linear process. These situations only serve to reinforce the need for an ordered process that can be understood and tracked even under adverse conditions.

Once a specific outline and UDF cover sheet have been established, it is imperative that the format and content of each section be clearly and completely defined as part of the project/company standards to avoid ambiguity and maintain consistency in the products. The following discussion expands and describes the contents of the UDF typified by the cover sheet shown in Figure 2.

#### Section 0. COVER SHEET AND SCHEDULE

This section contains the cover sheet for the unit, which identifies the routines included in the UDF and which delineates, for each of the sections, the scheduled due dates, actual completion dates, assigned originators and provides space for reviewer sign-offs and dates. In the case of multiple-routine units, it may be advisable to include a one-page composite schedule illustrating the section schedules of each item for easy check-off and monitoring. Following each cover sheet, a UDF Change Log should be included to document all UDF changes subsequent to the time when the initial development is completed and the unit is put into a controlled test or maintenance environment. Figure 3 illustrates a typical UDF Change Log.

#### Section 1. REQUIREMENTS

This section identifies the baseline requirements specification and enumerates the requirements which are allocated for implementation in the specific unit of software. A mapping to the system requirements specification (by paragraph number) should be made and, where practical, the statement of each requirement should be given. Any assumptions, ambiguities, deferrals or conflicts concerning the requirements and their impact on the design and development of the unit should be stated, and any design problem reports or deviations or waivers against the requirements should be indicated. In addition, if a requirement is only partially satisfied by this unit it will be so noted along with the unit(s) which share the responsibility for satisfaction of the requirement.

#### Section 2. DESIGN DESCRIPTION

This section contains the current design description for each of the routines included in the UDF. For multiple routine units, tabbed subsection separators are used for handy

UNIT DEVELOPMENT FOLDER COVER SHEET

PROGRAM NAME \_\_\_\_\_

UNIT NAME \_\_\_\_\_ CUSTODIAN \_\_\_\_\_

ROUTINES INCLUDED \_\_\_\_\_

SECTION NO.	DESCRIPTION	DUE DATE	DATE COMPLETED	ORIGINATOR	REVIEWER/ DATE
1	REQUIREMENTS				
2	DESIGN DESCRIPTION PRELIM: "CODE TO"				
3	FUNCTIONAL CAPABILITIES LIST				
4	UNIT CODE				
5	UNIT TEST PLAN				
6	TEST CASE RESULTS				
7	PROBLEM REPORTS				
8	NOTES				
9	REVIEWERS' COMMENTS				

- SECTION 1  
REQUIREMENTS
- SECTION 2  
DESIGN
- SECTION 3  
FCL
- SECTION 4  
UNIT CODE
- SECTION 5  
TEST PLAN
- SECTION 6  
TEST RESULTS
- SECTION 7  
PROBLEM REPORTS
- SECTION 8  
NOTES
- SECTION 9  
REVIEWERS' COMMENTS

Figure 2. UDF Cover Sheet and Layout

UDF CHANGE LOG

UNIT NAME \_\_\_\_\_ VERSION \_\_\_\_\_ CUSTODIAN \_\_\_\_\_

DATE	DPR/DJ Number	Section(s) Affected and Page Numbers	Retest Method	Mod No.

NOTE: This revision change log is to be used for all changes made in the UDF after internal baseline (i.e., subsequent to mod number assignment). It is inserted immediately after the coversheet.

Figure 3. Example of a UDF Change Log

indexing. A preliminary design description may be included if available; however, the end item for this section is detailed design documentation for the unit, suitable to become (part of) a "code to" specification. The format and content of this section should conform to established documentation standards and should be suitable for direct inclusion into the appropriate detailed design specification (Figure 1). Throughout the development process this section represents the current, working version of the design and, therefore, must be maintained and annotated as changes occur to the initial design. A flowchart is generally included as an inherent part of the design documentation. Flowcharts should be generated in accordance with clear established standards for content, format and symbol usage.

When the initial detailed design is completed and ready to be coded, a design walk-through may be held with one or more interested and knowledgeable co-workers. If such a walk-through is required, the completion of this section may be predicated on the successful completion of the design walk-through.

### Section 3. FUNCTIONAL CAPABILITIES LIST

This section contains a Functional Capabilities List (FCL) for the unit of software addressed by the UDF. An FCL is a list of the testable functions performed by the unit; i.e., it describes what a particular unit of software does, preferably in sequential order. The FCL is generated from the requirements and detailed design prior to development of the unit test plan. Its level of detail should correspond to the unit in question but, as a minimum, reflect the major segments of the code and the decisions which are being made. It is preferred that, whenever possible, functional capabilities be expressed in terms of the unit requirements (i.e., the functional capability is a requirement from Section 1 of the UDF). Requirements allocated to be tested at the unit level shall be included in the FCL. The FCL provides a vector from which the TEST CASE/REQUIREMENTS/FCL matrix (Figure 4) is generated. The FCL should be reviewed and addressed as part of the test plan review process.

The rationale for Functional Capabilities Lists is as follows:

- (1) They provide the basis for planned and controlled unit-level testing (i.e., a means for determining and organizing a set of test cases which will test all requirements/functional capabilities and all branches and transfers).
- (2) They provide a consistent approach to testing which can be reviewed, audited, and understood by an outsider. When mapped to the test cases, they provide the rationale for each test case.
- (3) They encourage another look at the design at a level where the "what if" questions can become apparent.

### Section 4. UNIT CODE

This section contains the current source code listings for each of the routines included in the unit. Indexed subsection separators are used for multiple routine units. The completion date for this section is the scheduled date for the first error-free compilation or assembly when the code is ready for unit-level testing. Where code listings or other



relevant computer output are too large or bulky to be contained in a normal three-ring binder, this material may be placed in a separate companion binder of appropriate size which is clearly identified with the associated UDF. In this event, the relevant sections of the UDF will contain a reference and identification of the binder with a history log of post-baselined updates. Figure 5 illustrates a typical reference form. 41

An independent review of the code may be optional; however, for time-critical or other technically important units, a code walk-through or review is recommended.

## Section 5. UNIT TEST PLAN

This section contains a description of the overall testing approach for the unit along with a description of each test case to be employed in testing the unit. The description must identify any test tools or drivers used, a listing of all required test inputs to the unit and their values, and the expected output and acceptance criteria, including numerical outputs and other demonstrable results. Test cases shall address the functional capabilities of the unit, and a matrix shall be placed into this section which correlates requirements and functional capabilities to test cases. This matrix will be used to demonstrate that all requirements, partial requirements, and FCLs of the unit have been tested. An example of the test case matrix is shown in Figure 4: Check marks are placed in the appropriate squares to correlate test cases with the capabilities tested. Sufficient detail should be provided in the test definition so that the test approach and objectives will be clear to an independent reviewer.

The primary criteria for the independent review will be to ascertain that the unit development test cases adequately test branch conditions, logic paths, input and output, error handling, a reasonable range of values and will perform as stipulated by the requirements. This review should occur prior to the start of unit testing.

## Section 6. TEST CASE RESULTS

This section contains a compilation of all current successful test case results and analyses necessary to demonstrate that the unit has been tested as described in the test plan. Test output should be identified by test case number and listings clearly annotated to facilitate necessary reviews of these results by other qualified individuals. Revision status of test drivers, test tools, data bases and unit code should be shown to facilitate retesting. This material may also be placed in the separate companion binder to the UDF.

## Section 7. PROBLEM REPORTS

This section contains status logs and copies of all Design Problem Reports, Design Analysis Reports and Discrepancy Reports (as required) which document all design and code problems and changes experienced by the unit subsequent to baselining. This ensures a clear and documented traceability for all problems and changes incurred. There should be separate subsections for each type of report with individual status logs that summarize the actions and dispositions made.



LISTINGS/TEST RESULTS

SEE SEPARATE NYLON PRONG BINDER IDENTIFIED AS \_\_\_\_\_ FOR CODE LISTINGS OR TEST RESULTS.

HISTORY LOG

CODE MOD NUMBER

DATE

REVIEWED BY

CODE MOD NUMBER	DATE	REVIEWED BY
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

Figure 5. Example Reference Log for Separately-Bound Material

## Section 8. NOTES

43

This section contains any memos, notes, reports, etc., which expand on the contents of the unit or are related to problems and issues involved.

## Section 9. REVIEWERS' COMMENTS

This section contains a record of reviewers' comments (if any) on this UDF, which have resulted from the section-by-section review and sign-off, and from scheduled independent audits. These reviewers' comments are also usually provided to the project and line management supervisors responsible for development of the unit.

## SUMMARY

44

The UDF concept has evolved into a practical, effective and valuable tool not only for the management of software development but also for imposing a structured approach on the total software development process. The structure and content of the UDF are designed to create a series of self-contained systems at the unit level, each of which can be easily observed and reviewed. The UDF approach has been employed on several software projects at TRW and continues to win converts from the ranks of the initiated. The concept has proved particularly effective when used in conjunction with good programming standards, documentation standards, a test discipline and an independent quality assurance activity.

The principal merits of the UDF concept are:

- (1) It imposes a development sequence on each unit and clearly establishes the responsibility for each step. Thus the reduction of the software development process into discrete activities is logically extended downward to the unit level.
- (2) It establishes a clearly-discernible timeline for the development of each unit and provides low-level management visibility into schedule problems. The status of the development effort becomes more visible and measurable.
- (3) It creates an open and auditable software development environment and removes some of the mystery often associated with this activity. The UDFs are normally kept "on the shelf" and open to inspection at any time.
- (4) It assures that the documentation is accomplished and maintained concurrent with development activities. The problem of emerging from the development tunnel with little or inadequate documentation is considerably reduced.
- (5) It reduces the problems associated with programmer turnover. The discipline and organization inherent in the approach simplifies the substitution of personnel at any point in the process without a significant loss of effort.
- (6) It supports the principles of modularity. The guidelines given for establishing the unit boundaries assure that at least a minimum level of modularity will result.
- (7) It can accommodate a variety of development plans and approaches. All UDF sections may be assigned to one performer, or different sections can be assigned to different specialists. The various sections contained in the UDF may also be expanded, contracted or even resequenced to better suit specific situations.

As a final comment, it must be emphasized that no device or approach can be effective without a strong management commitment to see it through. Every level of management needs to be supportive and aware of its responsibilities. Once the method is established it also needs to be audited for proper implementation and problem resolution. An independent software quality assurance activity can be a valuable asset in helping to define, audit and enforce management requirements.

# Applying the Technique of Configuration Management to Software

45

As the field of data processing continues to expand, the need to discipline the growth of computer programs (software) becomes more obvious. One management technique which promises to be effective is Configuration Management (C.M.). While this method was originally designed to control hardware production, its principles can be tailored and refined to relate to the development and production of computer software.

## The Growth of Software

In the last 10 years the field of computer software has expanded to make use of the greater speed and power of increasingly sophisticated computer hardware. Today high-level programming languages and complex operating systems are considered the norm. The natural path of growth has been to

more diverse applications, including:

- Large defense systems.
- Air traffic control systems.
- Medical software.

The complexity of the applications has also grown, resulting in software containing several hundred thousand lines of code.

The growth of the software industry has precipitated a rise in costs for software development, such that the expense of hardware is no longer the prime concern. For example, in 1971 the Air Force estimated their software expenses to be \$1-1.5 billion, which is about three times the expense of computer hardware.<sup>1</sup> The World Wide Military Command and Control System is estimated to cost \$42 to \$206 million for hardware and \$722 million for software.<sup>2</sup> Due to the cost of programming for larger and more complex software systems, software costs will continue to increase over

hardware costs, as shown in Figure 1<sup>3</sup> (see page 24).

In general, poor planning can be blamed for most of the software industry's inability to cope with these new demands and rapid growth. Projects were initiated without a clear goal; thus, as programmers coded, they made their own assumptions about the purpose of the programs and this often adversely affected the final product. Even when the goals were clearly specified, the development process suffered from inappropriate planning. Some managers and programmers measured progress in terms of the number of lines of code produced, and rushed to get something running as soon as possible. Auspicious beginnings proved misleading as unforeseen difficulties emerged. To solve the problem, much of the early code was rewritten, and eventually caused delays in delivery dates or the delivery of

by Rita McCarthy  
Burroughs Corporation  
Goleta, Calif.

Copyright © 1975 by the American Society for Quality Control, Inc. Reprinted by permission. No further reproduction authorized without permission of the Editor, *Quality Progress*, American Society for Quality Control, 161 West Wisconsin Avenue, Milwaukee, WI 53203.

<sup>1</sup> B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-49.

<sup>2</sup> Phil Hirsch, "GAO Hits Wimmie Hard; FY'72 Funding Prospects Fading Fast," *Datamation*, March 1, 1971, p. 41.

<sup>3</sup> B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-49.

incomplete products. Either alternative meant higher costs.

The final product was often characterized by a lack of reliability, which refers to the ability of a program to produce correct results when given a specific input. The consequences of such errors ranged from minor to disastrous. Minor problems do not have destructive side effects, but are often extremely annoying; for example, an incorrect page control on a printed report that causes a blank form on every other page. The failure of the Mariner I interplanetary probe, however, resulted from a disastrous error: the absence of one bar over a letter in a computational equation resulted in an unrecoverable problem, leaving no alternative but to destruct the \$18.5 million rocket shortly after launch.

The user of new software typically experienced very high error rates. Even after the first obvious errors were corrected and the product became operational, users continued to have a nearly constant pattern of failure. This was attributed to new errors introduced while correcting other problems and to the discovery of dormant errors as previously unused functions were tried. This phenomenon is discussed in an article by Jerry Ogden, and is represented in Figure 2. He also pointed out that modifications to already operational programs resulted in a rash of new failures, thus explaining the peak in the figure. While there is no single solution to all of these problems, the disciplines embodied in Configuration Management do provide a global framework in which specific solutions can be combined and monitored to attack specific parts of a problem.

## Defining Configuration Management

46

It is much easier to define Configuration Management by inspecting each word individually. "Configuration" applies to an interrelated group of programs that operate as a system. The term applies equally as well to the interrelating modules of one program. "Management" is the process of establishing and organizing objectives, followed by planning and employing resources to accomplish these objectives. The term "Configuration Management" is all-embracing, covering the management of

every detail of a software project from inception through development to completion and maintenance of the product.

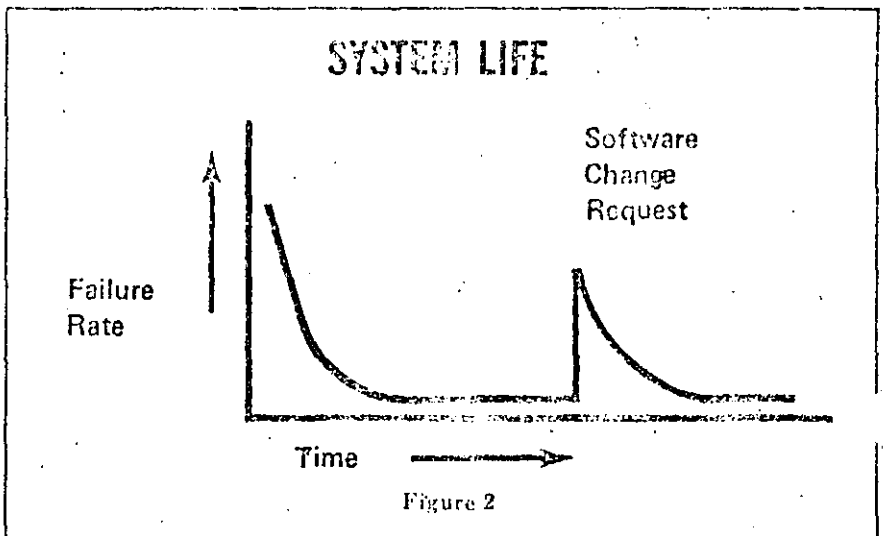
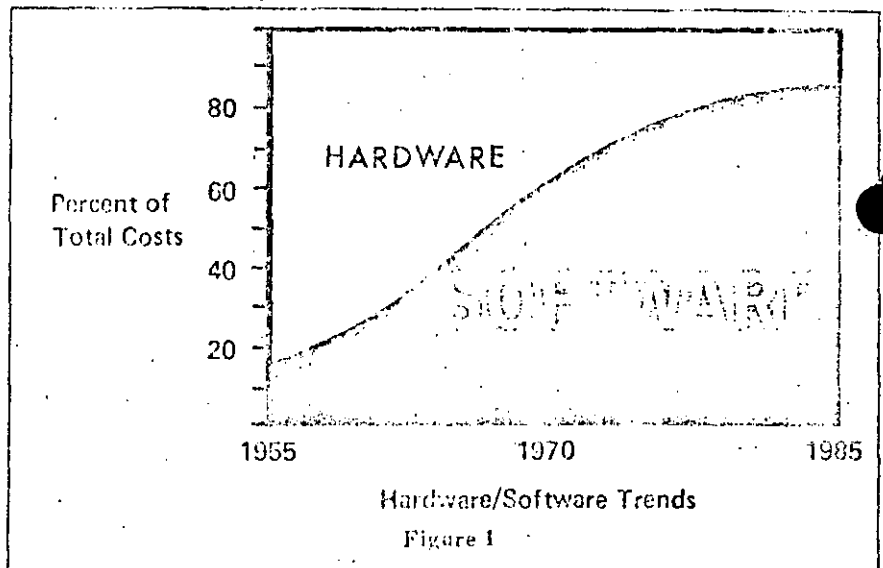
The objective of Configuration Management is to control the costs and the reliability of a software system. To achieve this, C.M. focuses on three areas:

- Identification.
- Control.
- Accounting.

The importance of these focal points is that they are directed

*J. L. Santer, "Reliability in Computer Programs," Mechanical Engineering, February 1969, p. 24.*

*Jerry L. Ogden, "Designing Reliable Software," Datamation, July 1972, pp. 71-78.*



ward all people involved with the product. C.M. attempts to combine the user, the administrators, the code developers, and the validation (test) team within the same framework, as shown in Figure 3.

### Identification

The philosophy of identification is to determine the exact nature of the problem, a suitable method of solution and the goals to guide the project before any actual coding begins. The assumption is that clear and complete information produces a cohesive, reliable product.

The identification process is concerned with the documentation of a design in progressively

finer levels of detail. This is accomplished by a series of reports which are individually described in the following.

**First Report.** An initial report on System Performance and Design Requirements must come from the customer and cover the following points:

- Definition of the desired product.
- Specific functions to be performed.
- Product environment, *e.g.*, hardware and operating systems, and limits on resources, *e.g.*, memory and storage media.
- Expected level of performance (speed).
- Reliability requirements (error tolerance).

### 47 Maintenance and support needs.

The first report is considered an important reliability tool. Features in the software that do not function according to expectations are commonly classified as errors. Therefore, it is imperative the customer be specific at this point, if the ultimate product is to be responsive to his needs. There should be no incomplete, conflicting, or uncertain terminology, which may lead to later problems in the software development process.

**Second Report.** As a reply to the initial report, the Part I Specification (Performance and Design Requirements for Computer Programs) is prepared. The de-

## Software Configuration Management

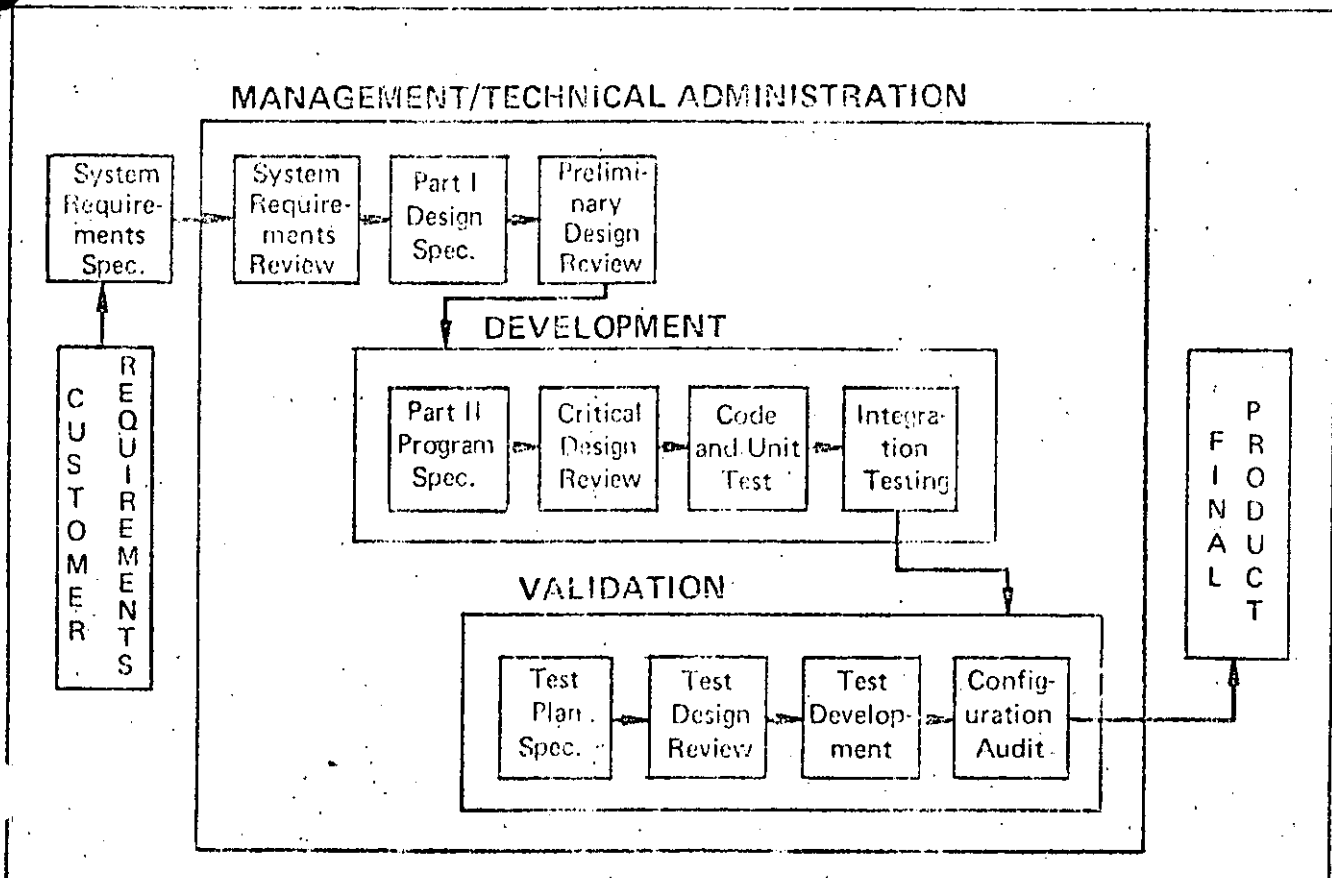


Figure 3

development people herein outline their method of solving the problem and their plan for ensuring reliability. Topics covered are discussed below.

- *General information flow.* This should include block diagrams showing input, processing and outputs indicating the sequence of events. There should be enough detail to provide the initial material for further program design.

- *Interface requirements.* An interface is a common boundary between parts of the system. For example, if one program accepts a file as input created by a previous program, the file becomes the path of their interface; errors can obviously occur at this point. Therefore, these interdependencies must be explicitly defined to assure that both programs are making the same assumptions about their interface. In some systems it may also be necessary to clarify the interface between the hardware and the software.

- *Expendability plan.* To plan a system which is easily modified and maintained, it is necessary to separate into independent areas those functions whose definitions are likely to change or expand. Thus, future modifications will be well isolated from other portions of the system and side-effect errors will not be introduced into already working code.

- *Test plan.* A test plan should be outlined for the development programmers and an independent validation (test) group. The development people must consider the testability of their design and ensure that code primitives can be exhaustively tested before the next higher level of code is added. Early location and correction of

errors results in a much more reliable program.

Specifications should be included for diagnostic tools that aid in the location of errors, e.g., execution time monitors or traces, and formatted memory or program dumps. Depending on the project, this list may be expanded to include hardware help such as readout displays. The development of these tools takes time, but it is more than recovered in assisting programmers to quickly and accurately locate their errors.

A solid test plan should provide for an independent validation team to be established at the beginning of the project. The responsibilities of this group are to follow the complete design of the product and independently specify, design and implement a comprehensive functional test library to be used in qualifying the final product before its release to the customer. The establishment of this independent group, which is less likely to make assumptions about the validity of the code, is a key step in assuring software reliability.

- *Reliability plan.* Programming standards or style to be imposed on all code should be outlined. A great deal has been learned recently about coding practices that increase program reliability (see Dijkstra<sup>6</sup> and Mills<sup>7</sup>). One proposed practice is called "Structured Programming," which involves dividing a complex program into progressively smaller modules, each of which has a well-defined task. The most refined modules are small and logically straightforward, have limited control structures and one entry and exit point, and are named by their function. The conciseness of the

modules allows the programmer to use formal mathematics to prove the correctness of the code (see Floyd<sup>8</sup> and London<sup>9</sup> for insights into the technique of proof of correctness).

While the primary intention of this second report is to interpret the problem and propose a solution to the customer, it also establishes an environment in which the solution can be achieved. Programmers and managers can consider testability, reliability and expandability on an equal priority with the process of coding.

**Third Report.** The Part II Specification (Product Specification for Computer Programs) is a complete and detailed technical description of the computer program(s) which describes how the solution to the problem is accomplished through the hierarchy of the code. Each refined module of the code is discussed. The details for each module include a description of its function, its expectations about global data and its effect on that data, and a description of its input and output. Such a design report essentially solves the entire programming problem in a narrative fashion before any coding begins. While this may be a trying experience for man-

<sup>6</sup> E. W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, Academic Press, New York, 1972.

<sup>7</sup> H. D. Mills, "Structured Programming in Large Systems," in *Debugging Techniques in Large Systems*, R. Rustin (ed.), Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

<sup>8</sup> R. W. Floyd, "Assigning Meaning to Programs," Proc. Symp. in Applied Mathematics, 19, J. T. Schwartz (ed.), American Math. Soc., Providence, Rhode Island, 1967, pp. 19-32.

<sup>9</sup> R. L. London, "Proving Programs Correct: Some Techniques and Examples," *BIT* Volume 10, 1970, p. 168.

agers accustomed to seeing lines of code and not technical documents, the results are encouraging.

The advantages of a good product design are twofold. First, it provides time for separating the complex problem into smaller, well-defined modules which are easier to understand, code, test and eventually modify. Secondly, it makes programmers confident of their approach, freeing them to concentrate on the accuracy of the code they write.

**Final Report.** The preparation of a Test Plan Specification should coincide with the preparation of the Part II Specification. This final report outlines the approach to be taken by the validation team in determining the functional accuracy and acceptable performance level of the software. The goal should be to provide for a test library of programs that are easy to operate, provide repeatable sequen-

ces of events, and are self-checking in nature. The detailed description of each test series includes: purpose, range of input data, expected output, priority and time required to do the test. A test plan that is complete in the coverage of the specified product can enhance confidence in the final product during the validation process.

**Control**

The control process is concerned with changes to be introduced into the software product. Because software is much more dynamic than hardware, there will always be new features to be incorporated, corrections to be made and code efficiency to be considered. Configuration control provides for these situations and establishes procedures for introducing proposals for change, evaluating these proposals, monitoring the status of the resulting action, and docu-

menting the effect of any change. Figure 4 is an example of a control procedure that incorporates several very worthwhile features:

- Requests can have variable origins.
- All requests follow the same steps.
- Management screens all requests to determine their impact on work loads.
- Requests and their status are maintained in a data base accessible to all.
- The data base can be used for reporting and accumulating statistics. It is possible to determine such things as the number and origin of errors reported, the amount of code changed during a given period of time, and the impact of these changes on other parts of the system.

The natural result of the control scheme is good communication. All levels of managers, de-

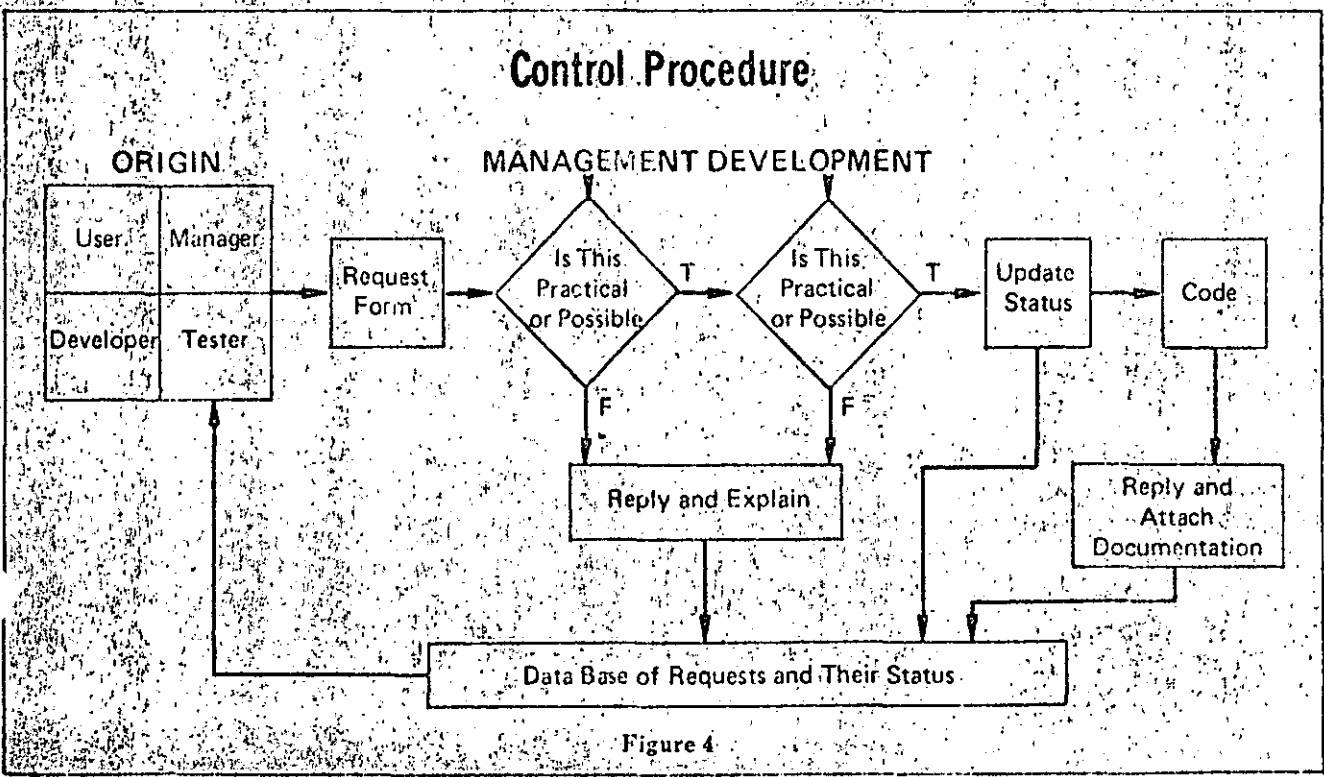


Figure 4



velopers, validators, and other personnel have a better chance of doing their work correctly and efficiently.

### Accounting

Configuration accounting provides continual visibility into software development through program reviews conducted at major milestones throughout the development, as well as through software documentation and product validation.

Formal program reviews are held for all involved with the software. The System Design Review covers the functional requirements of the system and the environment in which the product is to work. The Preliminary Design Review covers the overall design, plus the plans for expandability, testability and reliability. The Critical Design Review covers the technical details of each program of the system.

The usefulness of the reviews for each attendee varies depending on his position within the project. The development people are generally able to conceptualize their approach much better after a review. In summarizing the task, they are forced to re-examine all previous thought processes, and the strengths and weaknesses of the design become more apparent. The reviews are also useful for discussing the interfaces in software configurations of more than one program, assuring that the same interface procedure is being used with all of the programs.

During a review, managers gain insight into the progress of the project. An incomplete or disorganized presentation may reflect the actual state of affairs.

It may even be necessary to reschedule a review, requiring the development people to tie together loose ends. Participation in the reviews by the validation people helps them to better understand the product, enhancing the probability of thorough testing on their part and valid criticism of the generated documentation.

### Documentation

Management of the software documentation is another important aspect of configuration accounting. Without documentation, there is no history of the details, and it becomes difficult for programmers to compare approaches and verify interfaces. Furthermore, management has no visible sign of the progress of the programmers.

A part of the documentation related to the software is provided in the specification written during the identification procedure. This material describes the goals of the development effort and the technical details of the programs. Further, it is important to prepare a user's manual to describe the operational interface to the software system, enabling users to treat the software like a black box and ignore its internal workings.

The final point of accounting is also the last step of the development process: the Product Configuration Audit. Manuals, listings and programs (source and object) are delivered to the validation team, which has developed a complete test library paralleling the development of the actual software. The team is responsible for the quality assurance of the product and must determine if the software is per-

forming according to expectations: Change requests must be submitted for problems that are uncovered, and testing continues. If major problems occur, it may be necessary to suspend testing until the change requests are processed and new software is submitted.

The Product Configuration Audit measures the success of previous work. Therefore, complete records must be kept relating to the number of problems found and the progress made toward completion of the audit process. Determining the number and status of problems is relatively easy since this information is recorded in the data base of change requests.

Progress toward completing the audit is facilitated by a checklist of items to be tested. As each feature is verified, the date of checkout and the results are recorded. If there are problems, the feature is requalified after it is fixed by development, and the final checkout is recorded. When all tests are completed and the important problems are fixed, the software is delivered to the user with the user's manual and notification of any problems.

### Conclusion

The three essential requirements of identification, control and accounting provide a comprehensive base for a Configuration Management program, where details are flexibly tailored to meet the needs and goals of specific projects. The advantages to be accrued by Configuration Management are numerous, and any experience with its methods will help in facing the growing challenges for software in the future. □



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

SOFTWARE ENGINEERING ECONOMICS

DICIEMBRE, 1984

# Software Engineering Economics

BARRY W. BOEHM

1

**Abstract**—This paper summarizes the current state of the art and recent trends in software engineering economics. It provides an overview of economic analysis techniques and their applicability to software engineering and management. It surveys the field of software cost estimation, including the major estimation techniques available, the state of the art in algorithmic cost models, and the outstanding research issues in software cost estimation.

**Index Terms**—Computer programming costs, cost models, management decision aids, software cost estimation, software economics, software engineering, software management.

## I. INTRODUCTION

### Definitions

The dictionary defines "economics" as "a social science concerned chiefly with description and analysis of the production, distribution, and consumption of goods and services." Here is another definition of economics which I think is more helpful in explaining how economics relates to software engineering.

*Economics* is the study of how people make decisions in resource-limited situations.

This definition of economics fits the major branches of classical economics very well.

*Macroeconomics* is the study of how people make decisions in resource-limited situations on a national or global scale. It deals with the effects of decisions that national leaders make on such issues as tax rates, interest rates, foreign and trade policy.

*Microeconomics* is the study of how people make decisions in resource-limited situations on a more personal scale. It deals with the decisions that individuals and organizations make on such issues as how much insurance to buy, which word processor to buy, or what prices to charge for their products or services.

### Economics and Software Engineering Management

If we look at the discipline of software engineering, we see that the microeconomics branch of economics deals more with the types of decisions we need to make as software engineers or managers.

Clearly, we deal with limited resources. There is never enough time or money to cover all the good features we would like to put into our software products. And even in these days of cheap hardware and virtual memory, our more significant software products must always operate within a world of limited computer power and main memory. If you have been in the software engineering field for any length of time, I am sure

you can think of a number of decision situations in which you had to determine some key software product feature as a function of some limiting critical resource.

Throughout the software life cycle,<sup>1</sup> there are many decision situations involving limited resources in which software engineering economics techniques provide useful assistance. To provide a feel for the nature of these economic decision issues, an example is given below for each of the major phases in the software life cycle.

- *Feasibility Phase*: How much should we invest in information system analyses (user questionnaires and interviews, current-system analysis, workload characterizations, simulations, scenarios, prototypes) in order that we converge on an appropriate definition and concept of operation for the system we plan to implement?
- *Plans and Requirements Phase*: How rigorously should we specify requirements? How much should we invest in requirements validation activities (automated completeness, consistency, and traceability checks, analytic models, simulations, prototypes) before proceeding to design and develop a software system?
- *Product Design Phase*: Should we organize the software to make it possible to use a complex piece of existing software which generally but not completely meets our requirements?
- *Programming Phase*: Given a choice between three data storage and retrieval schemes which are primarily execution time-efficient, storage-efficient, and easy-to-modify, respectively; which of these should we choose to implement?
- *Integration and Test Phase*: How much testing and formal verification should we perform on a product before releasing it to users?
- *Maintenance Phase*: Given an extensive list of suggested product improvements, which ones should we implement first?
- *Phaseout*: Given an aging, hard-to-modify software product, should we replace it with a new product, restructure it, or leave it alone?

### Outline of This Paper

The economics field has evolved a number of techniques (cost-benefit analysis, present value analysis, risk analysis, etc.)

<sup>1</sup> Economic principles underlie the overall structure of the software life cycle, and its primary refinements of prototyping, incremental development, and advancement. The primary economic driver of the life-cycle structure is the significantly increasing cost of making a software change or fixing a software problem, as a function of the phase in which the change or fix is made. See [1], ch. 4].

Manuscript received April 26, 1983; revised June 28, 1983.  
The author is with the Software Information Systems Division, TRW Defense Systems Group, Redondo Beach, CA 90278.

### MASTER KEY TO SOFTWARE ENGINEERING ECONOMICS DECISION ANALYSIS TECHNIQUES

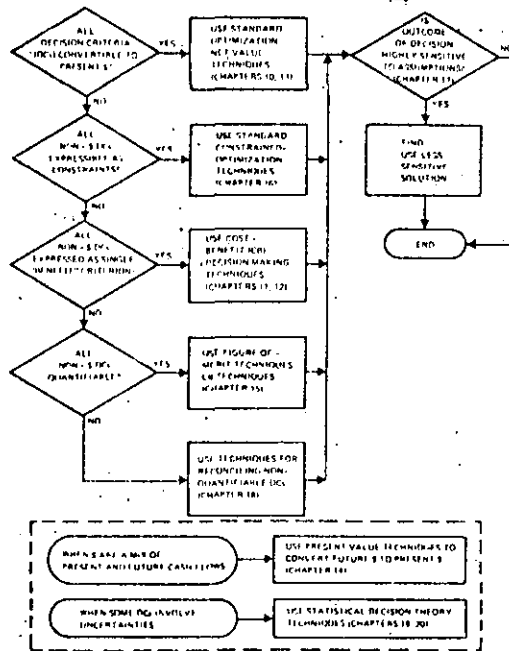


Fig. 1. Master key to software engineering economics decision analysis techniques.

for dealing with decision issues such as the ones above. Section II of this paper provides an overview of these techniques and their applicability to software engineering.

One critical problem which underlies all applications of economic techniques to software engineering is the problem of estimating software costs. Section III contains three major sections which summarize this field:

III-A: Major Software Cost Estimation Techniques

III-B: Algorithmic Models for Software Cost Estimation

III-C: Outstanding Research Issues in Software Cost Estimation.

Section IV concludes by summarizing the major benefits of software engineering economics, and commenting on the major challenges awaiting the field.

## II. SOFTWARE ENGINEERING ECONOMICS ANALYSIS TECHNIQUES

### Overview of Relevant Techniques

The microeconomics field provides a number of techniques for dealing with software life-cycle decision issues such as the ones given in the previous section. Fig. 1 presents an overall master key to these techniques and when to use them.<sup>2</sup>

<sup>2</sup> The chapter numbers in Fig. 1 refer to the chapters in [11], in which those techniques are discussed in further detail.

As indicated in Fig. 1, standard optimization techniques can be used when we can find a single quantity such as dollars (or pounds, yen, cruzeiros, etc.) to serve as a "universal solvent" into which all of our decision variables can be converted. Or, if the nondollar objectives can be expressed as constraints (system availability must be at least 98 percent; throughput must be at least 150 transactions per second), then standard constrained optimization techniques can be used. And if cash flows occur at different times, then present-value techniques can be used to normalize them to a common point in time.

More frequently, some of the resulting benefits from the software system are not expressible in dollars. In such situations, one alternative solution will not necessarily dominate another solution.

An example situation is shown in Fig. 2, which compares the cost and benefits (here, in terms of throughput in transactions per second) of two alternative approaches to developing an operating system for a transaction processing system.

- *Option A:* Accept an available operating system. This will require only \$80K in software costs, but will achieve a peak performance of 120 transactions per second, using five \$10K minicomputer processors, because of a high multiprocessor overhead factor.
- *Option B:* Build a new operating system. This system would be more efficient and would support a higher peak throughput, but would require \$180K in software costs.

The cost-versus-performance curve for these two options are shown in Fig. 2. Here, neither option dominates the other, and various cost-benefit decision-making techniques (maximum profit margin, cost/benefit ratio, return on investments, etc.) must be used to choose between Options A and B.

In general, software engineering decision problems are even more complex than Fig. 2, as Options A and B will have several important criteria on which they differ (e.g., robustness, ease of tuning, ease of change, functional capability). If these criteria are quantifiable, then some type of figure of merit can be defined to support a comparative analysis of the preferability of one option over another. If some of the criteria are unquantifiable (user goodwill, programmer morale, etc.), then some techniques for comparing unquantifiable criteria need to be used. As indicated in Fig. 1, techniques for each of these situations are available, and discussed in [11].

### Analyzing Risk, Uncertainty, and the Value of Information

In software engineering, our decision issues are generally even more complex than those discussed above. This is because the outcome of many of our options cannot be determined in advance. For example, building an operating system with a significantly lower multiprocessor overhead may be achievable, but on the other hand, it may not. In such circumstances, we are faced with a problem of *decision making under uncertainty*, with a considerable *risk* of an undesired outcome.

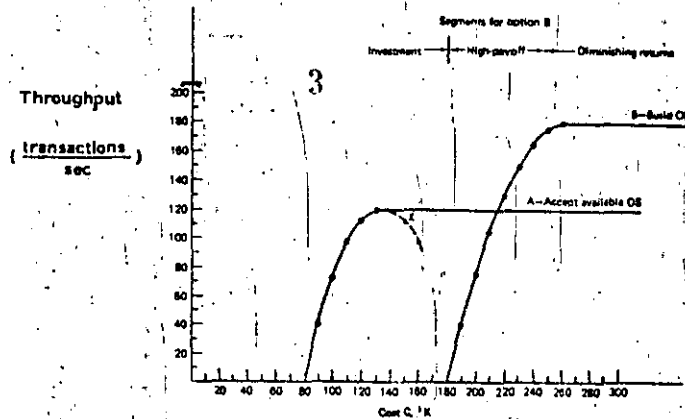


Fig. 2. Cost-effectiveness comparison, transaction processing system options.

The main economic analysis techniques available to support us in resolving such problems are the following.

1) Techniques for decision making under complete uncertainty, such as the maximax rule, the maximin rule, and the Laplace rule [38]. These techniques are generally inadequate for practical software engineering decisions.

2) Expected-value techniques, in which we estimate the probabilities of occurrence of each outcome (successful or unsuccessful development of the new operating system) and complete the expected payoff of each option:

$$EV = \text{Prob}(\text{success}) \cdot \text{Payoff}(\text{successful OS}) \\ + \text{Prob}(\text{failure}) \cdot \text{Payoff}(\text{unsuccessful OS}).$$

These techniques are better than decision making under complete uncertainty, but they still involve a great deal of risk if the  $\text{Prob}(\text{failure})$  is considerably higher than our estimate of it.

3) Techniques in which we reduce uncertainty by *buying information*. For example, *prototyping* is a way of buying information to reduce our uncertainty about the likely success or failure of a multiprocessor operating system; by developing a rapid prototype of its high-risk elements, we can get a clearer picture of our likelihood of successfully developing the full operating system.

In general, prototyping and other options for buying information<sup>3</sup> are most valuable aids for software engineering decisions. However, they always raise the following question: "how much information-buying is enough?"

In principle, this question can be answered via statistical decision theory techniques involving the use of Bayes' Law, which allows us to calculate the expected payoff from a software project as a function of our level of investment in a prototype

<sup>3</sup> Other examples of options for buying information to support software engineering decisions include feasibility studies, user surveys, simulation, testing, and mathematical program verification techniques.

or other information-buying option. (Some examples of the use of Bayes' Law to estimate the appropriate level of investment in a prototype are given in [11, ch. 20].)

In practice, the use of Bayes' Law involves the estimation of a number of conditional probabilities which are not easy to estimate accurately. However, the Bayes' Law approach can be translated into a number of *value-of-information guidelines*, or conditions under which it makes good sense to decide on investing in more information before committing ourselves to a particular course of action.

*Condition 1: There exist attractive alternatives whose payoff varies greatly, depending on some critical states of nature.* If not, we can commit ourselves to one of the attractive alternatives with no risk of significant loss.

*Condition 2: The critical states of nature have an appreciable probability of occurring.* If not, we can again commit ourselves without major risk. For situations with extremely high variations in payoff, the appreciable probability level is lower than in situations with smaller variations in payoff.

*Condition 3: The investigations have a high probability of accurately identifying the occurrence of the critical states of nature.* If not, the investigations will not do much to reduce our risk of loss due to making the wrong decision.

*Condition 4: The required cost and schedule of the investigations do not overly curtail their net value.* It does us little good to obtain results which cost more than they can save us, or which arrive too late to help us make a decision.

*Condition 5: There exist significant side benefits derived from performing the investigations.* Again, we may be able to justify an investigation solely on the basis of its value in training, team-building, customer relations, or design validation.

#### *Some Pitfalls Avoided by Using the Value-of-Information Approach*

The guideline conditions provided by the value-of-information approach provide us with a perspective which helps us avoid some serious software engineering pitfalls. The pitfalls

below are expressed in terms of some frequently expressed but faulty pieces of software engineering advice.

**Pitfall 1: Always use a simulation to investigate the feasibility of complex realtime software.** Simulations are often extremely valuable in such situations. However, there have been a good many simulations developed which were largely an expensive waste of effort, frequently under conditions that would have been picked up by the guidelines above. Some have been relatively useless because, once they were built, nobody could tell whether a given set of inputs was realistic or not (picked up by Condition 3). Some have been taken so long to develop that they produced their first results the week after the proposal was sent out, or after the key design review was completed (picked up by Condition 4).

**Pitfall 2: Always build the software twice.** The guidelines indicate that the prototype (or build-it-twice) approach is often valuable, but not in all situations. Some prototypes have been built of software whose aspects were all straightforward and familiar, in which case nothing much was learned by building them (picked up by Conditions 1 and 2).

**Pitfall 3: Build the software purely top-down.** When interpreted too literally, the top-down approach does not concern itself with the design of low level modules until the higher levels have been fully developed. If an adverse state of nature makes such a low level module (automatically forecast sales volume, automatically discriminate one type of aircraft from another) impossible to develop, the subsequent redesign will generally require the expensive rework of much of the higher level design and code. Conditions 1 and 2 warn us to temper our top-down approach with a thorough top-to-bottom software risk analysis during the requirements and product design phases.

**Pitfall 4: Every piece of code should be proved correct.** Correctness proving is still an expensive way to get information on the fault-freedom of software, although it strongly satisfies Condition 3 by giving a very high assurance of a program's correctness. Conditions 1 and 2 recommend that proof techniques be used in situations where the operational cost of a software fault is very large, that is, loss of life, compromised national security, major financial losses. But if the operational cost of a software fault is small, the added information on fault-freedom provided by the proof will not be worth the investment (Condition 4).

**Pitfall 5: Nominal-case testing is sufficient.** This pitfall is just the opposite of Pitfall 4. If the operational cost of potential software faults is large, it is highly imprudent not to perform off-nominal testing.

#### Summary: The Economic Value of Information

Let us step back a bit from these guidelines and pitfalls. Put simply, we are saying that, as software engineers:

"It is often worth paying for information because it helps us make better decisions."

If we look at the statement in a broader context, we can see that it is the primary reason why the software engineering field exists. It is what practically all of our software customers say when they decide to acquire one of our products: that it is worth paying for a management information system, a weather

forecasting system, an air traffic control system, an inventory control system, etc., because it helps them make better decisions.

Usually, software engineers are *producers* of management information to be consumed by other people, but during the software life cycle we must also be *consumers* of management information to support our own decisions. As we come to appreciate the factors which make it attractive for us to pay for processed information which helps us make better decisions as software engineers, we will get a better appreciation for what our customers and users are looking for in the information processing systems we develop for them.

### III. SOFTWARE COST ESTIMATION

#### Introduction

All of the software engineering economics decision analysis techniques discussed above are only as good as the input data we can provide for them. For software decisions, the most critical and difficult of these inputs to provide are estimates of the cost of a proposed software project. In this section, we will summarize:

- 1) the major software cost estimation techniques available, and their relative strengths and difficulties;
- 2) algorithmic models for software cost estimation;
- 3) outstanding research issues in software cost estimation.

#### A. Major Software Cost Estimation Techniques

Table I summarizes the relative strengths and difficulties of the major software cost estimation methods in use today.

1) **Algorithmic Models:** These methods provide one or more algorithms which produce a software cost estimate as a function of a number of variables which are considered to be the major cost drivers.

2) **Expert Judgment:** This method involves consulting one or more experts, perhaps with the aid of an expert-consensus mechanism such as the Delphi technique.

3) **Analogy:** This method involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project.

4) **Parkinson:** A Parkinson principle ("work expands to fill the available volume") is invoked to equate the cost estimate to the available resources.

5) **Price-to-Win:** Here, the cost estimate is equated to the price believed necessary to win the job (or the schedule believed necessary to be first in the market with a new product, etc.).

6) **Top-Down:** An overall cost estimate for the project is derived from global properties of the software product. The total cost is then split up among the various components.

7) **Bottom-Up:** Each component of the software job is separately estimated, and the results aggregated to produce an estimate for the overall job.

The main conclusions that we can draw from Table I are the following.

- None of the alternatives is better than the others from all aspects.
- The Parkinson and price-to-win methods are unacceptable and do not produce satisfactory cost estimates.

TABLE I  
STRENGTHS AND WEAKNESSES OF SOFTWARE  
COST ESTIMATION METHODS

5

Method	Strengths	Weaknesses
Algorithmic model	<ul style="list-style-type: none"> <li>Objective, repeatable, analyzable formula</li> <li>Efficient, good for sensitivity analysis</li> <li>Objectively calibrated to experience</li> </ul>	<ul style="list-style-type: none"> <li>Subjective inputs</li> <li>Assessment of exceptional circumstances</li> <li>Calibrated to past, not future</li> </ul>
Expert judgment	<ul style="list-style-type: none"> <li>Assessment of representativeness, interactions, exceptional circumstances</li> </ul>	<ul style="list-style-type: none"> <li>No better than participants</li> <li>Biases, incomplete recall</li> </ul>
Analogy	<ul style="list-style-type: none"> <li>Based on representative experience</li> </ul>	<ul style="list-style-type: none"> <li>Representativeness of experience</li> </ul>
Partneer	<ul style="list-style-type: none"> <li>Correlates with some experience</li> </ul>	<ul style="list-style-type: none"> <li>Reinforces poor practice</li> </ul>
Price to win	<ul style="list-style-type: none"> <li>Often gets the contract</li> </ul>	<ul style="list-style-type: none"> <li>Generally produces large overrun</li> </ul>
Top-down	<ul style="list-style-type: none"> <li>System level focus</li> <li>Efficient</li> </ul>	<ul style="list-style-type: none"> <li>Less detailed bases</li> <li>Less stable</li> </ul>
Bottom-up	<ul style="list-style-type: none"> <li>More detailed bases</li> <li>More stable</li> <li>Fosters individual commitment</li> </ul>	<ul style="list-style-type: none"> <li>May overlook system level costs</li> <li>Requires more effort</li> </ul>

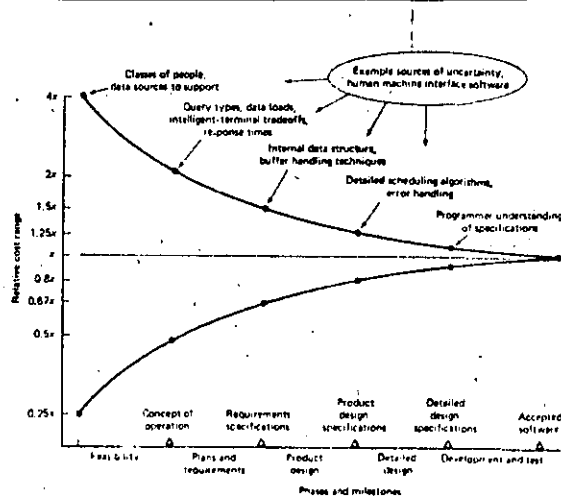


Fig. 3. Software cost estimation accuracy versus phase.

• The strengths and weaknesses of the other techniques are complementary (particularly the algorithmic models versus expert judgment and top-down versus bottom-up).

• Thus, in practice, we should use combinations of the above techniques, compare their results, and iterate on them where they differ.

#### Fundamental Limitations of Software Cost Estimation Techniques

Whatever the strengths of a software cost estimation technique, there is really no way we can expect the technique to compensate for our lack of definition or understanding of the software job to be done. Until a software specification is fully defined, it actually represents a range of software products, and a corresponding range of software development costs.

This fundamental limitation of software cost estimation technology is illustrated in Fig. 3, which shows the accuracy within which software cost estimates can be made, as a function of the software life-cycle phase (the horizontal axis), or of the level of knowledge we have of what the software is intended to do. This level of uncertainty is illustrated in Fig. 3

with respect to a human-machine interface component of the software.

When we first begin to evaluate alternative concepts for a new software application, the relative range of our software cost estimates is roughly a factor of four on either the high or low side.<sup>4</sup> This range stems from the wide range of uncertainty we have at this time about the actual nature of the product. For the human-machine interface component, for example, we do not know at this time what classes of people (clerks, computer specialists, middle managers, etc.) or what classes of data (raw or pre-edited, numerical or text, digital or analog) the system will have to support. Until we pin down such uncertainties, a factor of four in either direction is not surprising as a range of estimates.

The above uncertainties are indeed pinned down once we complete the feasibility phase and settle on a particular concept of operation. At this stage, the range of our estimates diminishes to a factor of two in either direction. This range is

<sup>4</sup> These ranges have been determined subjectively, and are intended to represent 80 percent confidence limits, that is, "within a factor of four on either side, 80 percent of the time."

reasonable because we still have not pinned down such issues as the specific types of user query to be supported, or the specific functions to be performed within the microprocessor in the intelligent terminal. These issues will be resolved by the time we have developed a software requirements specification, at which point, we will be able to estimate the software costs within a factor of 1.5 in either direction.

By the time we complete and validate a product design specification, we will have resolved such issues as the internal data structure of the software product and the specific techniques for handling the buffers between the terminal microprocessor and the central processors on one side, and between the microprocessor and the display driver on the other. At this point, our software estimate should be accurate to within a factor of 1.25, the discrepancies being caused by some remaining sources of uncertainty such as the specific algorithms to be used for task scheduling, error handling, abort processing, and the like. These will be resolved by the end of the detailed design phase, but there will still be a residual uncertainty about 10 percent based on how well the programmers really understand the specifications to which they are to code. (This factor also includes such consideration as personnel turnover uncertainties during the development and test phases.)

#### B. Algorithmic Models for Software Cost Estimation

##### Algorithmic Cost Models: Early Development

Since the earliest days of the software field, people have been trying to develop algorithmic models to estimate software costs. The earliest attempts were simple rules of thumb, such as:

- on a large project, each software performer will provide an average of one checked-out instruction per man-hour (or roughly 150 instructions per man-month);
- each software maintenance person can maintain four boxes of cards (a box of cards held 2000 cards, or roughly 2000 instructions in those days of few comment cards).

Somewhat later, some projects began collecting quantitative data on the effort involved in developing a software product, and its distribution across the software life cycle. One of the earliest of these analyses was documented in 1956 in [8]. It indicated that, for very large operational software products on the order of 100 000 delivered source instructions (100 KDSI), that the overall productivity was more like 64 DSI/man-month, that another 100 KDSI of support software would be required; that about 15 000 pages of documentation would be produced and 3000 hours of computer time consumed; and that the distribution of effort would be as follows:

Program Specs:	10 percent
Coding Specs:	30 percent
Coding:	10 percent
Parameter Testing:	20 percent
Assembly Testing:	30 percent

with an additional 30 percent required to produce operational specs for the system. Unfortunately, such data did not become well known, and many subsequent software projects went through a painful process of rediscovering them.

During the late 1950's and early 1960's, relatively little

progress was made in software cost estimation, while the frequency and magnitude of software cost overruns was becoming critical to many large systems employing computers. In 1964, the U.S. Air Force contracted with System Development Corporation for a landmark project in the software cost estimation field. This project collected 104 attributes of 169 software projects and treated them to extensive statistical analysis. One result was the 1965 SDC cost model [41] which was the best possible statistical 13-parameter linear estimation model for the sample data:

$$\begin{aligned}
 MM = & -33.63 \\
 & +9.15 \text{ (Lack of Requirements) (0-2)} \\
 & +10.73 \text{ (Stability of Design) (0-3)} \\
 & +0.51 \text{ (Percent Math Instructions)} \\
 & +0.46 \text{ (Percent Storage/Retrieval Instructions)} \\
 & +0.40 \text{ (Number of Subprograms)} \\
 & +7.28 \text{ (Programming Language) (0-1)} \\
 & -21.45 \text{ (Business Application) (0-1)} \\
 & +13.53 \text{ (Stand-Alone Program) (0-1)} \\
 & +12.35 \text{ (First Program on Computer) (0-1)} \\
 & +58.82 \text{ (Concurrent Hardware Development) (0-1)} \\
 & +30.61 \text{ (Random Access Device Used) (0-1)} \\
 & +29.55 \text{ (Difference Host, Target Hardware) (0-1)} \\
 & +0.54 \text{ (Number of Personnel Trips)} \\
 & -25.20 \text{ (Developed by Military Organization) (0-1)}.
 \end{aligned}$$

The numbers in parentheses refer to ratings to be made by the estimator.

When applied to its database of 169 projects, this model produced a mean estimate of 40 MM and a standard deviation of 62 MM; not a very accurate predictor. Further, the application of the model is counterintuitive; a project with all zero ratings is estimated at minus 33 MM; changing language from a higher order language to assembly language adds 7 MM, independent of project size. The most conclusive result from the SDC study was that there were too many nonlinear aspects of software development for a linear cost-estimation model to work very well.

Still, the SDC effort provided a valuable base of information and insight for cost estimation and future models. Its cumulative distribution of productivity for 169 projects was a valuable aid for producing or checking cost estimates. The estimation rules of thumb for various phases and activities have been very helpful, and the data have been a major foundation for some subsequent cost models.

In the late 1960's and early 1970's, a number of cost models were developed which worked reasonably well for a certain restricted range of projects to which they were calibrated. Some of the more notable examples of such models are those described in [3], [54], [57].

The essence of the TRW Wolverton model [57] is shown in Fig. 4, which shows a number of curves of software cost per object instruction as a function of relative degree of difficulty



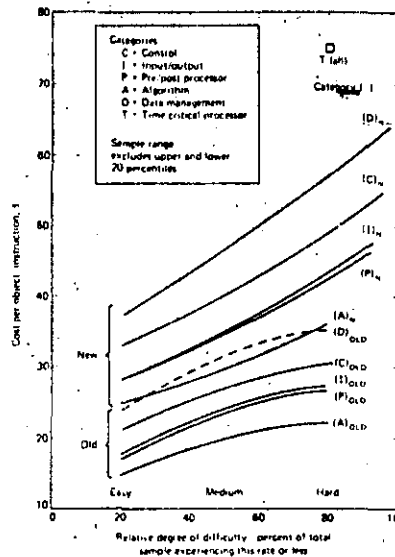


Fig. 4. TRW Wolverton model: Cost per object instruction versus relative degree of difficulty.

(0 to 100), novelty of the application (new or old), and type of project. The best use of the model involves breaking the software into components and estimating their cost individually. This, a 1000 object-instruction module of new data management software of medium (50 percent) difficulty would be costed at \$46/instruction, or \$46 000.

This model is well-calibrated to a class of near-real-time government command and control projects, but is less accurate for some other classes of projects. In addition, the model provides a good breakdown of project effort by phase and activity.

In the late 1970's, several software cost estimation models were developed which established a significant advance in the state of the art. These included the Putnam SLIM Model [44], the Doty Model [27], the RCA PRICE S model [22], the COCOMO model [11], the IBM-FSD model [53], the Boeing model [9], and a series of models developed by GRC [15]. A summary of these models, and the earlier SDC and Wolverton models, is shown in Table II, in terms of the size, program, computer, personnel, and project attributes used by each model to determine software costs. The first four of these models are discussed below.

#### The Putnam SLIM Model [44], [45]

The Putnam SLIM Model is a commercially available (from Quantitative Software Management, Inc.) software product based on Putnam's analysis of the software life cycle in terms of the Rayleigh distribution of project personnel level versus time. The basic effort macro-estimation model used in SLIM is

$$S_s = C_k K^{1/3} t_d^{4/3}$$

where

- $S_s$  = number of delivered source instructions
- $K$  = life-cycle effort in man-years
- $t_d$  = development time in years
- $C_k$  = a "technology constant."

Values of  $C_k$  typically range between 610 and 57 314. The current version of SLIM allows one to calibrate  $C_k$  to past projects or to past projects or to estimate it as a function of a project's use of modern programming practices, hardware constraints, personnel experience, interactive development, and other factors. The required development effort,  $DE$ , is estimated as roughly 40 percent of the life-cycle effort for large systems. For smaller systems, the percentage varies as a function of system size.

The SLIM model includes a number of useful extensions to estimate such quantities as manpower distribution, cash flow, major-milestone schedules, reliability levels, computer time, and documentation costs.

The most controversial aspect of the SLIM model is its tradeoff relationship between development effort  $K$  and between development time  $t_d$ . For a software product of a given size, the SLIM software equation above gives

$$K = \frac{\text{constant}}{t_d^4}$$

For example, this relationship says that one can cut the cost of a software project in half, simply by increasing its development time by 19 percent (e.g., from 10 months to 12 months). Fig. 5 shows how the SLIM tradeoff relationship com-

TABLE II  
FACTORS USED IN VARIOUS COST MODELS

GROUP	FACTOR	SDC, 1968	TW, 1972	PUTNAM, SLIM	DOTY	RCA, PRICE S	IBM	BOEING, 1977	GRC, 1979	COCOMO	SOFCOST	DSM	JENSEN
SIZE ATTRIBUTES	SOURCE INSTRUCTIONS			X	X	X	X	X		X	X	X	X
	OBJECT INSTRUCTIONS	X	X		X	X							
	NUMBER OF ROUTINES	X				X					X		
	NUMBER OF DATA ITEMS						X			X	X		
	NUMBER OF OUTPUT FORMATS								X			X	
	DOCUMENTATION				X		X	X			X		X
PROGRAM ATTRIBUTES	NUMBER OF PERSONNEL			X		X	X	X		X	X	X	X
	TYPE	X	X	X	X	X	X	X		X	X	X	X
	COMPLEXITY		X	X	X	X	X			X	X	X	X
	LANGUAGE	X		X				X	X		X	X	X
	REUSE			X		X		X	X		X	X	X
COMPUTER ATTRIBUTES	REQUIRED RELIABILITY			X		X			X	X	X	X	X
	DISPLAY REQUIREMENTS				X					X		X	
	TIME CONSTRAINT		X	X	X	X	X	X	X	X	X	X	X
	STORAGE CONSTRAINT			X	X	X	X		X	X	X	X	X
	HARDWARE CONFIGURATION	X				X					X	X	X
	CONCURRENT HARDWARE DEVELOPMENT	X			X	X	X			X	X	X	X
PERSONNEL ATTRIBUTES	INTERFACING EQUIPMENT, SAW									X	X	X	X
	PERSONNEL CAPABILITY			X		X	X			X	X	X	X
	PERSONNEL CONTINUITY					X	X				X	X	X
	HARDWARE EXPERIENCE	X		X	X	X	X		X	X	X	X	X
	APPLICATIONS EXPERIENCE		X	X	X	X	X	X	X	X	X	X	X
PROJECT ATTRIBUTES	LANGUAGE EXPERIENCE			X		X	X	X	X	X	X	X	X
	TOOLS AND TECHNIQUES			X		X	X	X		X	X	X	X
	CUSTOMER INTERFACE	X				X					X	X	X
	REQUIREMENTS DEFINITION	X			X	X					X	X	X
	REQUIREMENTS VOLATILITY	X			X	X			X	X	X	X	X
	SCHEDULE			X		X			X	X	X	X	X
	SECURITY					X					X	X	X
	COMPUTER ACCESS			X	X	X	X	X		X	X	X	X
	TRAVEL/REHOSTING/MULTI SITE	X			X	X					X	X	X
CALIBRATION FACTOR	SUPPORT SOFTWARE MATURITY								X		X	X	X
				X		X			X				
EFFORT EQUATION	$MM_{NOM} = C(DSH)^X$		1.0		1.047		0.91	1.0		1.05-1.2		1.0	1.2
SCHEDULE EQUATION	$T_D = C(MM)^X$						0.35			0.32-0.38		0.356	0.333

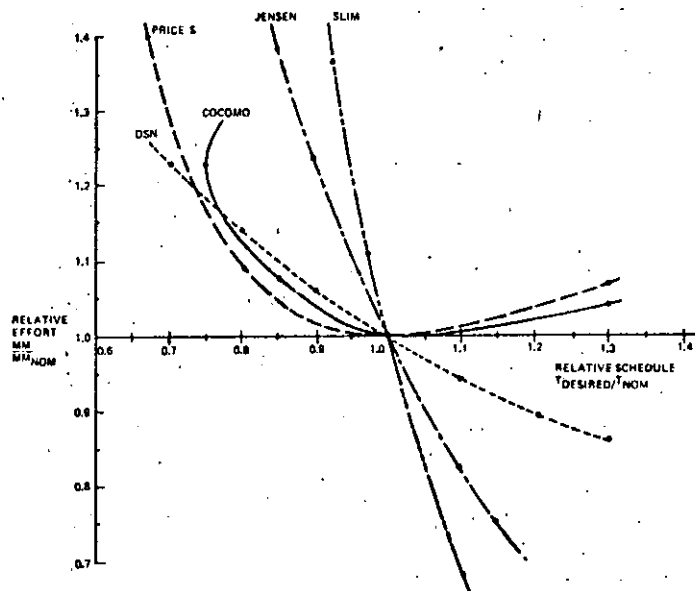


Fig. 5. Comparative effort-schedule tradeoff relationships.

TABLE III  
DOTY MODEL FOR SMALL PROGRAMS\*

$MM = 2.060 \left( \prod_{i=1}^{14} f_i \right)^{1.047}$  9

Factor	$f_i$	Yes	No
Special display	$f_1$	1.11	1.00
Detailed definition of operational requirements	$f_2$	1.00	1.11
Change to operational requirements	$f_3$	1.05	1.00
Real-time operation	$f_4$	1.33	1.00
CPU memory constraint	$f_5$	1.43	1.00
CPU time constraint	$f_6$	1.33	1.00
First software developed on CPU	$f_7$	1.92	1.00
Concurrent development of ADP hardware	$f_8$	1.82	1.00
Timeshare versus batch processing, in development	$f_9$	0.63	1.00
Developer using computer at another facility	$f_{10}$	1.43	1.00
Development at operational site	$f_{11}$	1.39	1.00
Development computer different than target computer	$f_{12}$	1.25	1.00
Development at more than one site	$f_{13}$	1.25	1.00
Programmer access to computer	$f_{14}$	Limited Unlimited	1.00 0.90

\* Less than 10,000 source instructions

compares with those of other models; see [11, ch. 27] for further discussion of this issue.

On balance, the SLIM approach has provided a number of useful insights into software cost estimation, such as the Rayleigh-curve distribution for one-shot software efforts, the explicit treatment of estimation risk and uncertainty, and the cube-root relationship defining the minimum development time achievable for a project requiring a given amount of effort.

#### The Doty Model [27]

This model is the result of an extensive data analysis activity, including many of the data points from the SDC sample. A number of models of similar form were developed for different application areas. As an example, the model for general application is

$$MM = 5.288 (KDSI)^{1.047}, \quad \text{for } KDSI > 10$$

$$MM = 2.060 (KDSI)^{1.047} \left( \prod_{i=1}^{14} f_i \right), \quad \text{for } KDSI < 10.$$

The effort multipliers  $f_i$  are shown in Table III. This model has a much more appropriate functional form than the SDC model, but it has some problems with stability, as it exhibits a discontinuity at  $KDSI = 10$ , and produces widely varying estimates via the  $f$  factors (answering "yes" to "first software developed on CPU" adds 92 percent to the estimated cost).

#### The RCA PRICE S Model [22]

PRICE S is a commercially available (from RCA, Inc.) macro cost-estimation model developed primarily for embedded system applications. It has improved steadily with experience; earlier versions with a widely varying subjective complexity factor have been replaced by versions in which a number of computer, personnel, and project attributes are used to modulate the complexity rating.

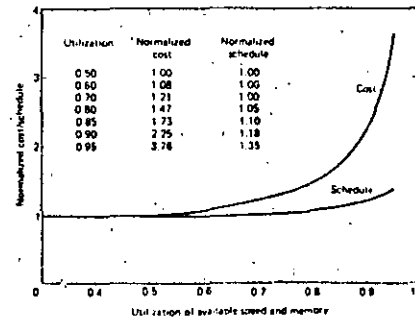


Fig. 6. RCA PRICE S model: Effect of hardware constraints.

PRICE S has extended a number of cost-estimating relationships developed in the early 1970's such as the hardware constraint function shown in Fig. 6 [10]. It was primarily developed to handle military software projects, but now also includes rating levels to cover business applications.

PRICE S also provides a wide range of useful outputs on gross phase and activity distributions analyses, and monthly project cost-schedule-expected progress forecasts. Price S uses a two-parameter beta distribution rather than a Rayleigh curve to calculate development effort distribution versus calendar time.

PRICE S has recently added a software life-cycle support cost estimation capability called PRICE SL [34]. It involves the definition of three categories of support activities.

- **Growth:** The estimator specifies the amount of code to be added to the product. PRICE SL then uses its standard techniques to estimate the resulting life-cycle-effort distribution.

- **Enhancement:** PRICE SL estimates the fraction of the existing product which will be modified (the estimator may

TABLE VIII  
COCOMO MODULE COMPLEXITY RATINGS VERSUS TYPE OF  
MODULE

10

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
Very low	Straightline code with a few non-nested SP <sup>2</sup> operators, DOs, CASEs, IFTHENELSEs. Simple predicates.	Evaluation of simple expressions: e.g., $A = B - C$ , $(D - E)$ .	Simple read, write statements with simple formats.	Simple arrays in main memory.
Low	Straightforward nesting of SP operators. Mostly simple predicates.	Evaluation of moderate-level expressions: e.g., $D = SORT(B^{**}2 - 4 * A * C)$ .	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap.	Single file subsetting with no data structure changes, no edits, no intermediate files.
Nominal	Mostly simple nesting. Some inter-module control. Decision tables.	Use of standard math and statistical routines. Basic matrix/vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits.
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable inter-module control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations, basic truncation, roundoff concerns.	Operations at physical I/O level (physical storage address translations, seeks, rcs, js, etc). Optimized I/O overlap.	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level.
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling.	Difficult but structured N.A., near-singular matrix equations, partial differential equations.	Routines for interrupt diagnosis, servicing, masking. Communication line handling.	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization.
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control.	Difficult and unstructured N.A.; highly accurate analysis of noisy, stochastic data.	Device timing-dependent coding, micro-programmed operations.	Highly coupled, dynamic relational structures. Natural language data management.

A SP = structured programming

TABLE IX  
COCOMO COST DRIVER RATINGS: MICROPROCESSOR  
COMMUNICATIONS SOFTWARE

Cost Driver	Situation	Rating	Effort Multiplier
RELY	Serious financial consequences of software faults	High	1.15
DATA	20,000 bytes	Low	0.99
CPLX	Communications processing	Very High	1.30
TIME	Will use 70% of available time	High	1.11
STOR	45K of 64K store (70%)	High	1.06
VIRT	Based on commercial microprocessor hardware	Nominal	1.00
TURN	Two-hour average turnaround time	Nominal	1.00
ACAP	Good senior analysts	High	0.86
AEXP	Three years	Nominal	1.00
PCAP	Good senior programmers	High	0.86
VEXP	Six months	Low	1.10
LEXP	Twelve months	Nominal	1.00
MODP	Most techniques in use over one year	High	0.91
TOOL	At basic minicomputer tool level	Low	1.10
SCED	Nine months	Nominal	1.00
	Effort adjustment factor (product of effort multipliers)		1.35

The effort multipliers for the other cost driver attributes are obtained similarly, except for the Complexity attribute, which is obtained via Table VIII. Here, we first determine that communications processing is best classified under device-dependent operations (column 3 in Table VIII). From this column, we determine that communication line handling typically has a complexity rating of Very High; from Table VI, then, we determine that its corresponding effort multiplier is 1.30.

**Step 3—Estimate Development Effort:** We then compute the estimated development effort for the microprocessor communications software as the nominal development effort (44 MM) times the product of the effort multipliers for the 15 cost driver attributes in Table IX (1.35, in Table IX). The resulting estimated effort for the project is then

$$(44 \text{ MM}) (1.35) = 59 \text{ MM.}$$

**Step 4—Estimate Related Project Factors:** COCOMO has additional cost estimating relationships for computing the resulting dollar cost of the project and for the breakdown of cost and effort by life-cycle phase (requirements, design, etc.) and by type of project activity (programming, test planning, management, etc.). Further relationships support the estimation of the project's schedule and its phase distribution. For example, the recommended development schedule can be obtained from the estimated development man-months via the embedded-mode schedule equation in Table V:

$$T_{DEV} = 2.5(59)^{0.32} = 9 \text{ months.}$$

As mentioned above, COCOMO also supports the most common types of sensitivity analysis and tradeoff analysis involved in scoping a software project. For example, from Tables VI and VII, we can see that providing the software developers with an interactive computer access capability (Low turnaround time) reduces the TURN effort multiplier from 1.00 to 0.87, and thus reduces the estimated project effort from 59 MM to

$$(59 \text{ MM}) (0.87) = 51 \text{ MM.}$$

The COCOMO model has been validated with respect to a sample of 63 projects representing a wide variety of business, scientific, systems, real-time, and support software projects. For this sample, Intermediate COCOMO estimates come within 20 percent of the actuals about 68 percent of the time (see Fig. 7). Since the residuals roughly follow a normal distribution, this is equivalent to a standard deviation of roughly 20 percent of the project actuals. This level of accuracy is representative of the current state of the art in software cost models. One can do somewhat better with the aid of a calibration coefficient (also a COCOMO option), or within a limited applications context, but it is difficult to improve significantly on this level of accuracy while the accuracy of software data collection remains in the "±20 percent" range.

A Pascal version of COCOMO is available for a nominal distribution charge from the Wang Institute, under the name WICOMO [18].

#### Recent Software Cost Estimation Models

Most of the recent software cost estimation models tend to follow the Doty and COCOMO models in having a nominal

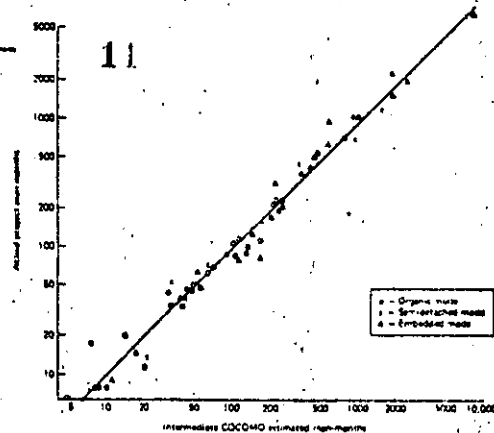


Fig. 7. Intermediate COCOMO estimates versus project actuals.

scaling equation of the form  $MM_{NOM} = c(KDSI)^x$  and a set of multiplicative effort adjustment factors determined by a number of cost driver attribute ratings. Some of them use the Rayleigh curve approach, to estimate distribution across the software life-cycle, but most use a more conservative effort/schedule tradeoff relation than the SLIM model. These aspects have been summarized for the various models in Table II and Fig. 5.

The Bailey-Basili meta-model [4] derived the scaling equation

$$MM_{NOM} = 3.5 + 0.73 (KDSI)^{1.16}$$

and used two additional cost driver attributes (methodology level and complexity) to model the development effort of 18 projects in the NASA-Goddard Software Engineering Laboratory to within a standard deviation of 15 percent. Its accuracy for other project situations has not been determined.

The Grumman SOFCOST Model [19] uses a similar but unpublished nominal effort scaling equation, modified by 30 multiplicative cost driver variables rated on a scale of 0 to 10. Table II includes a summary of these variables.

The Tausworthe Deep Space Network (DSN) model [50] uses a linear scaling equation ( $MM_{NOM} = a(KDSI)^{1.0}$ ) and a similar set of cost driver attributes, also summarized in Table II. It also has a well-considered approach for determining the equivalent KDSI involved in adapting existing software within a new product. It uses the Rayleigh curve to determine the phase distribution of effort, but uses a considerably more conservative version of the SLIM effort-schedule tradeoff relationship (see Fig. 5).

The Jensen model [30], [31] is a commercially available model with a similar nominal scaling equation, and a set of cost driver attributes very similar to the Doty and COCOMO models (but with different effort multiplier ranges); see Table II. Some of the multiplier ranges in the Jensen model vary as functions of other factors; e.g., increasing access to computer resources widens the multiplier ranges on such cost drivers as personnel capability and use of software tools. It uses the Rayleigh curve for effort distribution, and a somewhat more conservative ef-

fort-schedule tradeoff relation than SLIM (see Fig. 5). As with the other commercial models, the Jensen model produces a number of useful outputs on resource expenditure rates, probability distributions on costs and schedules, etc.

#### C. Outstanding Research Issues in Software Cost Estimation

Although a good deal of progress has been made in software cost estimation, a great deal remains to be done. This section updates the state-of-the-art review published in [11], and summarizes the outstanding issues needing further research:

- 1) Software size estimation;
- 2) Software size and complexity metrics;
- 3) Software cost driver attributes and their effects;
- 4) Software cost model analysis and refinement;
- 5) Quantitative models of software project dynamics;
- 6) Quantitative models of software life-cycle evolution;
- 7) Software data collection.

1) *Software Size Estimation*: The biggest difficulty in using today's algorithmic software cost models is the problem of providing sound sizing estimates. Virtually every model requires an estimate of the number of source or object instructions to be developed, and this is an extremely difficult quantity to determine in advance. It would be most useful to have some formula for determining the size of a software product in terms of quantities known early in the software life cycle, such as the number and/or size of the files, input formats, reports, displays; requirements specification elements, or design specification elements.

Some useful steps in this direction are the function-point approach in [2] and the sizing estimation model of [29], both of which have given reasonably good results for small-to-medium sized business programs within a single data processing organization. Another more general approach is given by DeMarco in [17]. It has the advantage of basing its sizing estimates on the properties of specifications developed in conformance with DeMarco's paradigm models for software specifications and designs: number of functional primitives, data elements, input elements, output elements, states, transitions between states, relations, modules, data tokens, control tokens, etc. To date, however, there has been relatively little calibration of the formulas to project data. A recent IBM study [14] shows some correlation between the number of variables defined in a state-machine design representation and the product size in source instructions.

Although some useful results can be obtained on the software sizing problem, one should not expect too much. A wide range of functionality can be implemented beneath any given specification element or I/O element, leading to a wide range of sizes (recall the uncertainty ranges of this nature in Fig. 3). For example, two experiments, involving the use of several teams developing a software program to the same overall functional specification, yielded size ranges of factors of 3 to 5 between programs (see Table X).

The primary implication of this situation for practical software sizing and cost estimation is that *there is no royal road to software sizing*. This is no magic formula that will provide an easy and accurate substitute for the process of thinking through and fully understanding the nature of the software product to be developed. There are still a number of useful

TABLE X  
SIZE RANGES OF SOFTWARE PRODUCTS PERFORMING SAME FUNCTION

Experiment	Product	No. of Teams	Size Range (source-instr.)
Weinberg & Schulman [55]	Simultaneous linear equations	6	33-165
Boehm, Gray, & Seewaldt [13]	Interactive cost model	7	1314-4606

things that one can do to improve the situation, including the following.

- Use techniques which explicitly recognize the ranges of variability in software sizing. The PERT estimation technique [56] is a good example.
- Understand the primary sources of bias in software sizing estimates. See [11, ch. 21].
- Develop and use a corporate memory on the nature and size of previous software products.

2) *Software Size and Complexity Metrics*: Delivered source instructions (DSI) can be faulted for being too low-level a metric for use in early sizing estimation. On the other hand, DSI can also be faulted for being too high-level a metric for precise software cost estimation. Various complexity metrics have been formulated to more accurately capture the relative information content of a program's instructions, such as the Halstead Software Science metrics [24], or to capture the relative control complexity of a program, such as the metrics formulated by McCabe in [39]. A number of variations of these metrics have been developed; a good recent survey of them is given in [26].

However, these metrics have yet to exhibit any practical superiority to DSI as a predictor of the relative effort required to develop software. Most recent studies [48], [32] show a reasonable correlation between these complexity metrics and development effort, but no better a correlation than that between DSI and development effort.

Further, the recent [25] analysis of the software science results indicates that many of the published software science "successes" were not as successful as they were previously considered. It indicates that much of the apparent agreement between software science formulas and project data was due to factors overlooked in the data analysis: inconsistent definitions and interpretations of software science quantities, unrealistic or inconsistent assumptions about the nature of the projects analyzed, overinterpretation of the significance of statistical measures such as the correlation coefficient, and lack of investigation of alternative explanations for the data. The software science use of psychological concepts such as the Stroud number have also been seriously questioned in [16].

The overall strengths and difficulties of software science are summarized in [47]. Despite the difficulties, some of the software science metrics have been useful in such areas as identifying error-prone modules. In general, there is a strong intuitive argument that more definitive complexity metrics will eventually serve as better bases for definitive software cost estimation than will DSI. Thus, the area continues to be an attractive one for further research.

3) *Software Cost Driver Attributes and Their Effects*: Most of the software cost models discussed above contain a selection of cost driver attributes and a set of coefficients, functions, or tables representing the effect of the attribute on software cost (see Table II). Chapters 24-28 of [11] contain summaries of the research to date on about 20 of the most significant cost driver attributes, plus statements of nearly 100 outstanding research issues in the area.

Since the publication of [11] in 1981, a few new results have appeared. Lawrence [35] provides an analysis of 278 business data processing programs which indicate a fairly uniform development rate in procedure lines of code per hour, some significant effects on programming rate due to batch turnaround time and level of experience, and relatively little effect due to use of interactive operation and modern programming practices (due, perhaps, to the relatively repetitive nature of the software jobs sampled). Okada and Azuma [42] analyzed 30 CAD/CAM programs and found some significant effects due to type of software, complexity, personnel skill level, and requirements volatility.

4) *Software Cost Model Analysis and Refinement*: The most useful comparative analysis of software cost models to date is the Thibodeau [52] study performed for the U.S. Air Force. This study compared the results of several models (the Wolverton, Doty, PRICE S, and SLIM models discussed earlier, plus models from the Boeing, SDC, Tecolote, and Aerospace corporations) with respect to 45 project data points from three sources.

Some generally useful comparative results were obtained, but the results were not definitive, as models were evaluated with respect to larger and smaller subsets of the data. Not too surprisingly, the best results were generally obtained using models with calibration coefficients against data sets with few points. In general, the study concluded that the models with calibration coefficients achieved better results, but that none of the models evaluated were sufficiently accurate to be used as a definitive Air Force software cost estimation model.

Some further comparative analyses are currently being conducted by various organizations, using the database of 63 software projects in [11], but to date none of these have been published.

In general, such evaluations play a useful role in model refinement. As certain models are found to be inaccurate in certain situations, efforts are made to determine the causes, and to refine the model to eliminate the sources of inaccuracy.

Relatively less activity has been devoted to the formulation, evaluation, and refinement of models to cover the effects of more advanced methods of software development (prototyping, incremental development, use of application generators, etc.) or to estimate other software-related life-cycle costs (conversion, maintenance, installation, training, etc.). An exception is the excellent work on software conversion cost estimation performed by the Federal Conversion Support Center [28]. An extensive model to estimate avionics software support costs using a weighted-multiplier technique has recently been developed [49]. Also, some initial experimental results have been obtained on the quantitative impact of prototyping in [13] and on the impact of very high level nonprocedural lan-

guages in [58]. In both studies, projects using prototyping and VHLL's were completed with significantly less effort.

5) *Quantitative Models of Software Project Dynamics*: Current software cost estimation models are limited in their ability to represent the internal dynamics of a software project, and to estimate how the project's phase distribution of effort and schedule will be affected by environmental or project management factors. For example, it would be valuable to have a model which would accurately predict the effort and schedule distribution effects of investing in more thorough design verification, of pursuing an incremental development strategy, of varying the staffing rate or experience mix, of reducing module size, etc.

Some current models assume a universal effort distribution, such as the Rayleigh curve [44] or the activity distributions in [57], which are assumed to hold for any type of project situation. Somewhat more realistic, but still limited are models with phase-sensitive effort multipliers such as PRICE S [22] and Detailed COCOMO [11].

Recently, some more realistic models of software project dynamics have begun to appear, although to date none of them have been calibrated to software project data. The Phister phase-by-phase model in [43] estimates the effort and schedule required to design, code, and test a software product as a function of such variables as the staffing level during each phase, the size of the average module to be developed, and such factors as interpersonal communications overhead rates and error detection rates. The Abdel Hamid-Madnick model [1], based on Forrester's System Dynamics world-view, estimates the time distribution of effort, schedule, and residual defects as a function of such factors as staffing rates, experience mix, training rates, personnel turnover, defect introduction rates, and initial estimation errors. Tausworthe [51] derives and calibrates alternative versions of the SLIM effort-schedule tradeoff relationship, using an intercommunication-overhead model of project dynamics. Some other recent models of software project dynamics are the Mitre SWAP model and the Duclos [21] total software life-cycle model.

6) *Quantitative Models of Software Life-Cycle Evolution*: Although most of the software effort is devoted to the software maintenance (or life-cycle support) phase, only a few significant results have been obtained to date in formulating quantitative models of the software life-cycle evolution process. Some basic studies by Belady and Lehman analyzed data on several projects and derived a set of fairly general "laws of program evolution" [7], [37]. For example, the first of these laws states:

"A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a re-created version."

Some general quantitative support for these laws was obtained in several studies during the 1970's, and in more recent studies such as [33]. However, efforts to refine these general laws into a set of testable hypotheses have met with mixed results. For

example, the Lawrence [36] statistical analysis of the Belady-Lahman data showed that the data supported an even stronger form of the first law ("systems grow in size over their useful life"); that one of the laws could not be formulated precisely enough to be tested by the data; and that the other three laws did not lead to hypotheses that were supported by the data.

However, it is likely that variant hypotheses can be found that are supported by the data (for example, the operating system data supports some of the hypotheses better than does the applications data). Further research is needed to clarify this important area.

7) *Software Data Collection*: A fundamental limitation to significant progress in software cost estimation is the lack of unambiguous, widely-used standard definitions for software data. For example, if an organization reports its "software development man-months," do these include the effort devoted to requirements analysis, to training, to secretaries, to quality assurance, to technical writers, to uncompensated overtime? Depending on one's interpretations, one can easily cause variations of over 20 percent (and often over a factor of 2) in the meaning of reported "software development man-months" between organizations (and similarly for "delivered instructions," "complexity," "storage constraint," etc.) Given such uncertainties in the ground data, it is not surprising that software cost estimation models cannot do much better than "within 20 percent of the actuals, 70 percent of the time."

Some progress towards clear software data definitions has been made. The IBM FSD database used in [53] was carefully collected using thorough data definitions, but the detailed data and definitions are not generally available. The NASA-Goddard Software Engineering Laboratory database [5], [6], [40] and the COCOMO database [11] provide both clear data definitions and an associated project database which are available for general use (and reasonably compatible). The recent Mitre SARE report [59] provides a good set of data definitions.

But there is still no commitment across organizations to establish and use a set of clear and uniform software data definitions. Until this happens, our progress in developing more precise software cost estimation methods will be severely limited.

#### IV. SOFTWARE ENGINEERING ECONOMICS BENEFITS AND CHALLENGES

This final section summarizes the benefits to software engineering and software management provided by a software engineering economics perspective in general and by software cost estimation technology in particular. It concludes with some observations on the major challenges awaiting the field.

##### *Benefits of a Software Engineering Economics Perspective*

The major benefit of an economic perspective on software engineering is that it provides a balanced view of candidate software engineering solutions, and an evaluation framework which takes account not only of the programming aspects of a situation, but also of the human problems of providing the best possible information processing service within a resource-limited environment. Thus, for example, the software engineering economics approach does not say, "we should use

these structured structures because they are mathematically elegant" or "because they run like the wind" or "because they are part of the structured revolution." Instead, it says "we should use these structured structures because they provide people with more benefits in relation to their costs than do other approaches." And besides the framework, of course, it also provides the techniques which help us to arrive at this conclusion.

##### *Benefits of Software Cost Estimation Technology*

The major benefit of a good software cost estimation model is that it provides a clear and consistent universe of discourse within which to address a good many of the software engineering issues which arise throughout the software life cycle. It can help people get together to discuss such issues as the following.

- Which and how many features should we put into the software product?
- Which features should we put in first?
- How much hardware should we acquire to support the software product's development, operation, and maintenance?
- How much money and how much calendar time should we allow for software development?
- How much of the product should we adapt from existing software?
- How much should we invest in tools and training?

Further, a well-defined software cost estimation model can help avoid the frequent misinterpretations, underestimates, overexpectations, and outright buy-ins which still plague the software field: In a good cost-estimation model, there is no way of reducing the estimated software cost without changing some objectively verifiable property of the software project. This does not make it impossible to create an unachievable buy-in, but it significantly raises the threshold of credibility.

A related benefit of software cost estimation technology is that it provides a powerful set of insights on how a software organization can improve its productivity. Many of a software cost model's cost-driver attributes are management controllables: use of software tools and modern programming practices, personnel capability and experience, available computer speed, memory, and turnaround time, software reuse. The cost model helps us determine how to adjust these management controllables to increase productivity, and further provides an estimate of how much of a productivity increase we are likely to achieve with a given level of investment. For more information on this topic, see [11, ch. 33], [12] and the recent plan for the U.S. Department of Defense Software Initiative [20].

Finally, software cost estimation technology provides an absolutely essential foundation for software project planning and control. Unless a software project has clear definitions of its key milestones and realistic estimates of the time and money it will take to achieve them, there is no way that a project manager can tell whether his project is under control or not. A good set of cost and schedule estimates can provide realistic data for the PERT charts, work breakdown structures, manpower schedules, earned value increments, etc., necessary to establish management visibility and control.

Note that this opportunity to improve management visibility and control requires a complementary management com-



mitment to define and control the reporting of data on software progress and expenditures. The resulting data are therefore worth collecting simply for their management value in comparing plans versus achievements, but they can serve another valuable function as well: they provide a continuing stream of calibration data for evolving a more accurate and refined software cost estimation models.

#### Software Engineering Economics Challenges

The opportunity to improve software project management decision making through improved software cost estimation, planning, data collection, and control brings us back full-circle to the original objectives of software engineering economics: to provide a better quantitative understanding of how software people make decisions in resource-limited situations.

The more clearly we as software engineers can understand the quantitative and economic aspects of our decision situations, the more quickly we can progress from a pure seat-of-the-pants approach on software decisions to a more rational approach which puts all of the human and economic decision variables into clear perspective. Once these decision situations are more clearly illuminated, we can then study them in more detail to address the deeper challenge: achieving a quantitative understanding of how people work together in the software engineering process.

Given the rather scattered and imprecise data currently available in the software engineering field, it is remarkable how much progress has been made on the software cost estimation problem so far. But, there is not much further we can go until better data becomes available. The software field cannot hope to have its Kepler or its Newton until it has had its army of Tycho Brahes, carefully preparing the well-defined observational data from which a deeper set of scientific insights may be derived.

#### REFERENCES

- [1] T. K. Abdel-Hamid and S. E. Madnick, "A model of software project management dynamics," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 539-554.
- [2] A. J. Albrecht, "Measuring Application Development Productivity," in *SHARE-GUIDE*, 1979, pp. 83-92.
- [3] J. D. Aron, "Estimating resources for large programming systems," NATO Sci. Committee, Rome, Italy, Oct. 1969.
- [4] J. J. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng., IEEE/IACM/NBS*, Mar. 1981, pp. 107-116.
- [5] V. R. Basili, "Tutorial on models and metrics for software and engineering," *IEEE Cat. EHO-167-7*, Oct. 1980.
- [6] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *Univ. Maryland Technol. Rep. TR-1235*, Dec. 1982.
- [7] L. A. Belady and M. M. Lehman, "Characteristics of large systems," in *Research Directions in Software Technology*, P. Wegner, Ed., Cambridge, MA: MIT Press, 1979.
- [8] H. D. Benington, "Production of large computer programs," in *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15-27.
- [9] R. K. D. Black, R. P. Curnow, R. Katz, and M. D. Gray, "BCS software production data," Boeing Comput. Services, Inc., Final Tech. Rep., RADC-TR-77-116, NTIS AD-A039852, Mar. 1977.
- [10] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, pp. 48-59, May 1973.
- [11] ———, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [12] B. W. Boehm, J. F. Elwell, A. B. Pyster, E. D. Stuckie, and R. D. Williams, "The TRW software productivity system," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982.
- [13] B. W. Boehm, T. E. Gray, and T. Scowald, "Prototyping vs. specifying: A multi-project experiment," *IEEE Trans. Software Eng.*, to be published.
- [14] R. N. Britcher and J. E. Gaffney, "Estimates of software size from state machine designs," in *Proc. NASA-Goddard Software Eng. Workshop*, Dec. 1982.
- [15] W. M. Carriere and R. Thibodeau, "Development of a logistics software cost estimating technique for foreign military sales," General Res. Corp., Rep. CR-3-839, June 1979.
- [16] N. S. Coulter, "Software science and cognitive psychology," *IEEE Trans. Software Eng.*, pp. 166-171, Mar. 1983.
- [17] T. DeMarco, *Controlling Software Projects*. New York: Yourdon, 1982.
- [18] M. Demshki, D. Ligett, B. Linn, G. McCluskey, and R. Miller, "Wang Institute cost model (WICOMO) tool user's manual," Wang Inst. Graduate Studies, Tyngsboro, MA, June 1982.
- [19] H. F. Dircks, "SOF-COST: Grumman's software cost eliminating model," in *IEEE NAECON 1981*, May 1981.
- [20] L. E. Druffel, "Strategy for DoD software initiative," RADC/DACS, Griffiss AFB, NY, Oct. 1982.
- [21] L. C. Duclous, "Simulation model for the life-cycle of a software product: A quality assurance approach," Ph.D. dissertation, Dep. Industrial and Syst. Eng., Univ. Southern California, Dec. 1982.
- [22] F. R. Freiman and R. D. Park, "PRICE software model—Version 3: An overview," in *Proc. IEEE-PINY Workshop on Quantitative Software Models*, *IEEE Cat. TH0067-9*, Oct. 1979, pp. 32-41.
- [23] R. Goldberg and H. Lorin, *The Economics of Information Processing*. New York: Wiley, 1982.
- [24] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
- [25] P. G. Hamer and G. D. Frewin, "M. H. Halstead's software science—A critical examination," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982, pp. 197-205.
- [26] W. Harrison, K. Magel, R. Kluczney, and A. DeKöck, "Applying software complexity metrics to program maintenance," *Computer*, pp. 65-79, Sept. 1982.
- [27] J. R. Herd, J. N. Postak, W. E. Russell, and K. R. Stewart, "Software cost estimation study—Study results," Doty Associates, Inc., Rockville, MD, Final Tech. Rep. RADC-TR-77-220, vol. 1 (of two), June 1977.
- [28] C. Houtz and T. Buschbach, "Review and analysis of conversion cost-estimating techniques," GSA Federal Conversion Support Center, Falls Church, VA, Rep. GSA/FCSC-81/001, Mar. 1981.
- [29] M. Itakura and A. Takayanagi, "A model for estimating program size and its evaluation," in *Proc. IEEE 6th Software Eng.*, Sept. 1982, pp. 104-109.
- [30] R. W. Jensen, "An improved macrolevel software development resource estimation model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 88-92.
- [31] R. W. Jensen and S. Lucas, "Sensitivity analysis of the Jensen software model," in *Proc. 5th ISPA Conf.*, Apr. 1983, pp. 384-389.
- [32] B. A. Kitchenham, "Measures of programming complexity," *ICL Tech. J.*, pp. 298-316, May 1981.
- [33] ———, "Systems evolution dynamics of VME/B," *ICL Tech. J.*, pp. 43-57, May 1982.
- [34] W. W. Kuhn, "A software lifecycle case study using the PRICE model," in *Proc. IEEE NAECON*, May 1982.
- [35] M. J. Lawrence, "Programming methodology, organizational environment, and programming productivity," *J. Syst. Software*, pp. 257-270, Sept. 1981.
- [36] ———, "An examination of evolution dynamics," in *Proc. IEEE 6th Int. Conf. Software Eng.*, Sept. 1982, pp. 188-196.
- [37] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, pp. 1060-1076, Sept. 1980.
- [38] R. D. Luce and H. Raiffa, *Games and Decisions*. New York: Wiley, 1957.
- [39] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, pp. 308-320, Dec. 1976.
- [40] F. E. McGarry, "Measuring software development technology: What have we learned in six years," in *Proc. NASA-Goddard Software Eng. Workshop*, Dec. 1982.
- [41] E. A. Nelson, "Management handbook for the estimation of computer programming costs," Syst. Develop. Corp., AD-A648750, Oct. 31, 1966.
- [42] M. Okada and M. Azuma, "Software development estimation study—A model from CAD/CAM, system development experiences," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 555-564.

- [43] M. Phister, Jr., "A model of the software development process," *J. Syst. Software*, pp. 237-256, Sept. 1981.
- [44] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, pp. 345-361, July 1978.
- [45] L. H. Putnam and A. Fitzsimmons, "Estimating software costs," *Datamation*, pp. 189-198, Sept. 1979; continued in *Datamation*, pp. 171-178, Oct. 1979, and pp. 137-140, Nov. 1979.
- [46] L. H. Putnam, "The real economics of software development," in *The Economics of Information Processing*, R. Goldberg and H. Lorin, New York: Wiley, 1982.
- [47] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, pp. 155-165, Mar. 1983.
- [48] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program complexity measure for software development management," in *Proc. IEEE 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 106-106.
- [49] SYSCON Corp., "Avionics software support cost model," USAF Avionics Lab., AFWAL-TR-1173, Feb. 1, 1983.
- [50] R. C. Tausworthe, "Deep space network software cost estimation model," Jet Propulsion Lab., Pasadena, CA, 1981.
- [51] —, "Staffing implications of software productivity models," in *Proc. 7th Annu. Software Eng. Workshop*, NASA/Goddard, Greenbelt, MD, Dec. 1982.
- [52] R. Thibodeau, "An evaluation of software cost estimating models," General Res. Corp., Rep. T10-2670, Apr. 1981.
- [53] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.
- [54] G. F. Weinwurm, Ed., *On the Management of Computer Programming*. New York: Auerbach, 1970.
- [55] G. M. Weinberg and E. L. Schulman, "Goals and performance in computer programming," *Human Factors*, vol. 16, no. 1, pp. 70-77, 1974.
- [56] J. D. Wiest and F. K. Levy, *A Management Guide to PERT/CPM*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
- [57] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, pp. 615-636, June 1974.
- [58] E. Harel and E. R. McLean, "The effects of using a nonprocedural computer language on programmer productivity," UCLA Inform. Sci. Working Paper 3-83, Nov. 1982.
- [59] R. L. Dunias, "Final report: Software acquisition resource expenditure (SARE) data collection methodology," MITRE Corp., MTR 9031, Sept. 1983.



Barry W. Boehm received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1957 and the M.A. and Ph.D. degrees from the University of California, Los Angeles, in 1961 and 1964, respectively.

From 1978 to 1979 he was a Visiting Professor of Computer Science at the University of Southern California. He is currently a Visiting Professor at the University of California, Los Angeles, and Chief Engineer of TRW's Software Information Systems Division. He was previously Head of the Information Sciences Department at The Rand Corporation, and Director of the 1971 Air Force CCIP-85 study. His responsibilities at TRW include direction of TRW's internal software R&D program, of contract software technology projects, of the TRW software development policy and standards program, of the TRW Software Cost Methodology Program, and the TRW Software Productivity Program. His most recent book is *Software Engineering Economics*, by Prentice-Hall.

Dr. Boehm is a member of the IEEE Computer Society and the Association for Computing Machinery, and an Associate Fellow of the American Institute of Aeronautics and Astronautics.

# Software Development Management Planning

JACK COOPER

17

**Abstract**—The lack of comprehensive planning prior to the initiation of a software development project is a very pervasive failing. This paper walks through a sample software development plan discussing the various areas that a software development manager should address in preparing his project's plan. Various considerations and suggestions are presented for each of the management subject areas. How the user/customer can use the developer's plan to aid in monitoring of his software's evolution is also presented. Detailed planning of a software development project is necessary to the successful completion of the project.

**Index Terms**—Project management, project planning, software acquisition management, software development plan, software engineering management.

- 1) Introduction
- 2) User Furnished Information, Equipment, Services, and Facilities
- 3) Risk Areas
- 4) Software Engineering Standards, Practices, and Procedures
- 5) Project Organization
- 6) Schedules and Milestones
- 7) Design Approach
- 8) Implementation Approach
- 9) Software Integration and Test
- 10) Software Development Facilities
- 11) Software Quality Assurance
- 12) Software Change Management
- 13) Product Turn-Over

Fig. 1. Sample software development plan (SDP) outline.

## I. INTRODUCTION

A VERY important phase in the life cycle of any automated system is the development of its software. Software development planning is crucial to the success of the project. This is especially true where the software is to be developed under contract. Planning of any project is a basic management procedure, but, it is surprising how seldom it is actually done to a sufficient degree.

The requisite planning takes two different forms depending on the manager's role in the development. If he is representing the User<sup>1</sup> of the system, he must establish a Software Life Cycle Management Plan that encompasses, not only the software development phase, but all other life cycle phases of the software as well. During software development he is concerned with enhancing the probability that the software to be delivered satisfies the original requirement and delivers on schedule at the lowest cost.

On the other hand, if he is the manager of the software development project, he must produce a Software Development Plan (SDP) (see Fig. 1) that provides for the management of all facets of the software engineering and development process. He is concerned that he is able to deliver the product software at a fair profit. For the remainder of this paper, software development planning will be viewed as it relates to the software developer.

A software development plan provides the comprehensive plan for the management of the software development effort. The SDP includes a description of the development organization, the technical approach, the milestones and schedules, and the allocation of resources. It provides the developer

with the means to coordinate schedules, control resources, initiate actions, and monitor progress of the development effort. Additionally, the SDP provides the user with the detailed knowledge of the schedule, organization, and resource allocation planned by the developer. It is one of the basic tools that the user uses in monitoring the work effort.

The SDP is produced by the development manager at the beginning of the project. As soon as the SDP is completed, an extremely valuable management ploy is to submit it to the user's Project Manager for his review and concurrence (do not forget to get his signature of approval). This tactic serves to commit both project managers, development and user, to jointly agreeing to the execution of the plan over the life of the development project. This will avert many disagreements further down the road as to how the project is supposed to be run. This is especially significant when one or more of the involved principals have changed. Once the SDP has been approved by the User-Project Manager, the developer then executes it.

In the case of contracts for major software developments, more and more, the request for proposal is asking for a preliminary version of the SDP to be included in the offerers' technical proposals. This serves the valuable purpose of allowing the user to assess, as a part of the selection process, the various offerers' approach to managing a software development project. One of the first tasks in the contract is to require the winning contractor to update and finalize his SDP by incorporating the comments that the user generated during the proposal evaluation and selection process. Having completed the SDP update, it is once again submitted to the User for his final approval. Once approved by User, it then becomes binding on the developer to implement and faithfully adhere to the plan throughout the life of the contract.

The next section will use the structure of a generic SDP, as depicted in Fig. 1, to discuss the various elements of management planning for a software development project.

Manuscript received January 20, 1983.

The author is with CACI, Inc., 1815 North Fort Meyer Drive, Arlington, VA 22209.

<sup>1</sup> For purposes of this article the terms User, customer, and acquisition manager, are synonymous. "User" will be used herein as a generic title for this category of persons. Also, this article will consider "contractor" as being synonymous with the generic term "developer."

## II. THE SOFTWARE DEVELOPMENT PLAN

### A. Introduction

18

The introduction section of the SDP should be constructed in such a manner that it may serve the additional purpose of an executive overview of the project. The background and history of the project should be presented followed by a description of the system for which the software is being developed. The purpose and scope of the software development effort, and the authority for the project should be officially promulgated. This section should include an overview of the management philosophy and methodology. Specific terms (e.g., program, module, routine, team leader, program analyst, parameter test, etc.) used in describing the development effort need to be defined in this section of the SDP. The terminology and definitions should be consistent with user imposed standards, whenever they exist, otherwise with the developing organization's internal standards. Project unique terminology and/or definitions serve as a significant point of confusion and should not be allowed.

### B. User Furnished Information, Equipment, Services, and Facilities

Prior to project start-up, the software development manager needs to research the information, equipment, services, and facilities that are to be furnished by the user for the production or test of the software. In addition to taking these items into consideration during project planning, they must also be brought to the attention of the user so that he can make the necessary arrangements in sufficient time so as not to impede the software development process. The schedule of usage and the training/maintenance support required, etc. must also be included.

### C. Risk Areas

Another critical action to be taken before proceeding blissfully down the software development road is to identify all of the potential risk areas. Every project has potential risk and problems; it would be extremely naive for a project manager to think that through his skillful ability all problems could be dealt with successfully as they occur in real time. Planning will, by its nature, tend to minimize the impact of the problems when they do occur. Areas to be researched include cost, schedule, requirements definition, technological implementation, and various types of security. Many of the high risk areas should be included on the project's critical path. Finally, this section of planning should also include provisions for alternative courses of action for each of the potential problems identified.

An important item to keep in mind while developing the SDP is that of constantly trying to avoid *unnecessary* complexity. There is a direct correlation between complexity and increased risk, and that correlates directly with increased time and costs. Keep things as simple as possible. Hardly is there ever a requirement in a software development project's charter to always use the latest and fanciest techniques, tools, practices, etc. Quite the contrary, charters typically fail to mention that subject and focus their attention on getting the job done within the time and resources allocated.

### D. Software Engineering Standards, Practices, and Procedures

A must for every SDP is a section that contains, or makes reference to the document that contains, the software engineering standards, practices, procedures, and conventions that will apply throughout the life of the software development project. These items must either be defined by the user or be agreed to by him, since he will have to live with them throughout the entire operations and maintenance phase of his software's life cycle.

In every software development there are many tradeoffs that will have to be made. They confront every person involved in the project from the top of the corporation down to the newest programmer trainee. It is important that these tradeoffs be made according to some uniform criteria rather than be made ad hoc in real time. If the user is alert, he will impose criteria on the software developer in the areas that are important to him. In the final analysis, he is the one most affected by the results of the tradeoff decisions. This area of the SDP is a convenient place to include the guidance for making all of the tradeoffs.

Extremely important to the success of any software development project is the attitude and approach taken toward computer program documentation. To treat computer program documentation casually is to invite disaster. A documentation standard that addresses, not only the design but the entire software development process, must be promulgated by the developer if the user fails to specify one. Procedures must be adopted to ensure the concurrent development of documents along with the code. Reverse engineering after-the-fact documentation is very expensive and time consuming. Quality assurance and change control of the documents must be provided. An informal, day-to-day documentation tool, such as the unit development folder or programmer's workbook, should be adopted. That type of tool is probably of even more value as a management aid.

The software developer should plan to issue a programming standards and conventions manual to all persons involved in the project prior to their beginning any work related to development of the software. This manual is the sheet of music that must guide all of their engineering activities. Hopefully, an internal standard document can be tailored to fit the unique aspects of this project. If not, then a new one will have to be developed. In either case, this document must be compatible with the long range software maintenance requirements of the user.

### E. Project Organization

First, the project's relationship to other organizational entities within the developer's company needs to be clearly stated. This should include the scope of each entity's authority as well as an itemization of all their responsibilities.

The description of the development project organization should next to be identified. All relevant job titles within the project and their interrelationships need to be established. Each job position must be described in sufficient detail to provide visibility and understanding of the project management structure. In addition to the narrative descriptions, graphic illustrations of the organization should be included.

In establishing the organization for the project there are some common mistakes that can be avoided. First, ensure the independence of the persons responsible for conducting the tests of the software. The testers must be objective in the conduct of their tests and they must be free of any pressures to overlook minor (or major) problems detected, to accept corrections that are less than satisfactory, or to cut short their required testing.

Also the independence of the group to be assigned the responsibility for the software quality assurance tasks must be ensured. They need an organizational reporting channel that is independent of the software development manager. When a project is in extremis, software quality assurance personnel must be shielded against any potential coercion by the project manager to lower their standards as an exigency of the circumstances. To do so may be an appropriate business decision, but that decision should be overt rather than covert. Objectivity is just as essential in quality assurance as it is in test.

Project Librarians are necessary to managing the changes to the software and for maintaining custody of the various products of the project. The concept of Team Librarians is becoming widely adopted as a cost effective method of optimizing the time of all team members, introducing entry level personnel safely and effectively into the project, providing an avenue for upwards mobility, and for extending the change control and product custody functions to the team level.

Finally, don't overlook training. There are a wide variety of training requirements in every software development project, not only during project start-up but also continuously over the life of the project.

#### *F. Schedules and Milestones*

Management planning is not complete without the inclusion of a schedule that depicts all activities and events that are to occur within the software development project. Significant milestones, critical paths, and potential critical paths which may occur due to unplanned schedule slippage must be included.

A very effective way to develop the schedule is to approach the task the same way you would approach the top-down design of a software system. Start at the top of the project and decompose it into its first line of major tasks. Then successively iterate the process decomposing the tasks into smaller, more comprehensive subtasks until the level is reached where the tasks cannot be subdivided any further. You will end up at the 50/100 lines of code level, etc. This procedure will result in a much larger and more detailed schedule than is customary. In very large projects it may be necessary, during project planning, to stop the process at some reasonable level prior to reaching the bottom. Once the project gets underway, the process can be completed by the line managers as a part of their planning.

In either case, there are several rewards for the effort. It avoids having to resort to percentages in status tracking, since tasks are small enough that they can be considered either not started or completed (0 or 100 complete). These small tasks also permit the allocation of resources, the loading of manpower, and the subsequent costing with a great deal of accuracy. All of which should be reflected in the project schedule.

In establishing the software development plan, beware of concepts that are based on the old "classical" software development model. This is especially important if those concepts are incompatible with the newer "top-down" software development model. Most susceptible is the area of the software development phases and overlapping of the phases.

#### *G. Design Approach*

A widely accepted rule of thumb states that 40 percent of the total software development effort is expended just getting to the point where coding begins, and that another 40 percent is expended once debug (or unit test) has been completed. That leaves only 20 percent of the total effort for coding and debug. Keep this rule in mind for the next three sections; the same proportions should be devoted to the software development planning effort.

To start off, plan for changes in requirements to occur throughout the complete development cycle. It is a fact of life that many requirements changes are necessary. For example, the way of doing business may change, hence, a system that is under development must also change in order to support the new procedures. Requirements changes are well known trouble makers; the trick is to recognize the fact that they will occur, plan for them, accommodate only the necessary ones, design a flexible system, and then control them when they do occur. Do not try to accommodate any of them as they occur in real time.

Design planning should identify the methods and techniques to be used to ensure the software design satisfies all technical, operational, and performance requirements. Remember that at this juncture planning for the design activities is being addressed, not "how to" actually create the design. A design approach, such as top-down, should be decided upon. Next, all of the methodologies needed to support that approach should be specified. Types of methodologies to be considered are some form of: structured requirements analysis, structured design, program design language, a flow-diagram-like graphic representation, simulation, modeling, and automatic documentation generation.

Design planning should include provisions for controlling and monitoring the utilization of system resources, such as memory, processor time, and I/O capacity. A certain amount reserve of system resources availability must be designed in up front if they are expected to exist at the end of the project. Recovery, through optimization, may not be successful. The user must have some reserve to carry into the maintenance phase of the software's life cycle for the correction of residual discrepancies and for minor enhancements.

#### *H. Implementation Approach*

The design planning must be extended into the planning for the implementation of the software. Most of those methods and techniques will continue to be applicable in the implementation phase. For example, the top-down software development methodology, the analysis and design tools, the computer program documentation development procedures, and the unit development folders, cannot be cost effectively discontinued or replaced. The continuous monitoring of the availability of the system resource reserves that were provided for in the design must now be put in place.

Coding techniques and production methods to be used (e.g., structured programming, programming teams, and team librarians) must now be identified. The concurrent development of the documentation must be ensured. Measures need to be implemented that will ensure that all software and documentation developed will be of the highest quality. Changes to all items being produced will be occurring on a daily basis, rules have to be instituted as to when changes may be made and what procedures must be followed in the process.

#### *I. Software Integration and Test*

Planning for the integration and test phase must be comprehensive and must be accomplished very early in the project. There could be required items of hardware or support software, such as a simulator, that are necessary to establish the appropriate test environment and whose acquisition might require a long lead time. This is especially true in projects with installed or furnished hardware with an operating system and where the top-down methodology is being used. This, through the use of stubs, permits software integration and testing to get an early start, which is one of the primary benefits of the methodology, and it could be lost due to a lack of early planning.

All system components (both hardware and software) need to be identified. Any applicable special simulation and test facilities, and how they will be used, must be taken into account. Test tools of all types will be required. The use of test drivers, simulators, data extraction and reduction, and other test support software and hardware, as applicable, need to be described. The schedule for usage and control over all of the physical facilities must be programmed or serious contention problems will continuously hamper the effort.

In addition to the requirements for the physical facilities, the approach, plans, and organization necessary to accomplish the integration and test all of the software to be delivered must be planned. The plan must contain a description of the integration sequence for all software components. The integration milestones should be shown on the project schedule and their relationships with the pertinent software component development milestones, including the required readiness of software components for integration.

The hardest decision a project manager has to make during the whole project is how often/when to recompile and relink the complete computer program system. Then, once the program is rebuilt, how much regression testing is to be conducted. This is a particularly difficult decision during the eleventh hour while trying to complete performance testing and obtain buy-off by the user. The SDP should address this subject and provide as much guidance as possible for the benefit on all project personnel. The tactic mentioned in the Introduction regarding getting the user's approval of the SDP will have a big pay-off when this issue does eventually arise. By his approval of the SDP, the user is also approving the recompile/regression test decision criteria.

A common testing pitfall is that of using the developing programmers to contribute in some way to the software system test effort. They are precisely the worst persons of all for this purpose. Once his debugging has been completed, the programmer's role should be limited to supporting the discrepancy correction activity. The primary purpose of testing

is to assess how accurately the completed software satisfies the original requirement, e.g., did the programmer go astray through misinterpretation of some requirement. Programmers characteristically will provide test data and test cases to test the programs that they wrote, i.e., the as-built--not the as required! Besides that, a different expertise is necessary, testers need to be able to exercise a system in the same way that the user will once the system is delivered.

#### *J. Software Development Facilities*

The capabilities of the software development and test facility should be published for the benefit of all project personnel. The projected usage throughout the development process needs to be estimated and included in the project master schedule. Project planning should address how they will be manned, and what special tools or facilities will be available during the software implementation process. Also, the management methods that will be used in the facility (e.g., programmer debug time, software change control and status accounting, quality assurance monitoring) need to be described.

#### *K. Software Quality Assurance (SQA)*

Software development management planning next needs to describe the policy, organization, and procedures to be used to ensure that the software to be delivered will comply with all of its specified requirements and is of high quality. SQA should be oriented toward ensuring that the type of design and implementation will result in effective and reliable software. SQA planning provides the developer with the means to monitor the quality assurance program as it is applied to the software development project. It provides the user with detailed knowledge of the developer's quality assurance program and it may be used to monitor the SQA program as it is being implemented.

First, the organization of the group responsible for the software quality assurance requirements needs to be described. The SDP should include a chart showing the relationship of the SQA group to management and other organizational entities. The general authority and responsibility of the SQA group must be formally promulgated.

This section of the SDP should identify all SQA policy, rules, techniques, and methodologies applicable in each area listed in Fig. 2, and describe how their use will augment or satisfy the SQA requirements.

#### *L. Software Change Control (SCC)*

SCC planning provides the developer means to consolidate all policies, procedures, organizational descriptions, resources, and schedules relating to software change control into one section of the SDP. It provides the user with detailed knowledge of the developer's software change control activities and it permits the user to monitor the developer's application of change control principles in conformance with requirements.

It is critical that the SCC section provide for the definition and establishment of the developmental baseline. Then, to institute a Software Change Control Board (SCCB) to adjudicate all proposed changes to that baseline. It must be specified when and under what conditions all software and documentation will come under formal change control. Also, to be in-

Software Development Management  
 Computer Program Documentation  
 Software Requirements  
 Testability Analysis  
 Computer Program Design  
 Interfaces of all Types  
 Database Definition  
 Software Implementation Process  
 Test Plans  
 Repeatability of Tests  
 Test Requirements and Criteria  
 Test Procedures  
 Test Report Certification  
 Corrective Actions  
 Trend Analysis  
 Reporting and Control System  
 Software Change Control  
 Quality Auditing

Fig. 2. Software quality assurance procedures areas.

cluded are instructions for the preparation, processing, and submittal of the proposed changes to the SCCB, including changes to the computer programs, documentation, and the contract/tasking agreement.

The SCC section must provide for procedures to ensure that the implementation of all approved changes is reflected in all facets of the baseline, program descriptive documentation, and program materials. Do not overlook the implementation and promulgation of all approved corrections to the software and documents.

Lastly, the SCC section of the SDP should provide for reconciliation of the software change status accounting reports and the status of the software, descriptive documentation, and program materials with the approved baseline, plus all approved changes.

#### M. Product Turn-Over

During development of the original SDP is not too early to address the planning and procedures for the orderly transition

21

of all software, its attendant documentation, and all other products from custody of the developer to that of the user. The discussion contained in Section I regarding long lead-time items is especially important in the case of any new facilities and/or support software required to maintain the software after completion of the software development process.

### III. CONCLUSION

If the foregoing sounds obvious, then accept the following challenge: using this article as a check-off list, conduct a review of your project to see how well it was originally planned and how current those plans are today!



Jack Cooper received the B.S. degree in agriculture from the University of Missouri, Columbia, and the M.S. degree in computer science for the Naval Post Graduate School, Monterey, CA.

He is currently Manager of the Software Acquisition Management Department of CACI, Inc., Arlington, VA. In this capacity he is primarily responsible for all software related business areas associated with embedded computer systems. Previous to this, he spent four years as President of Anchor Software Management, Ltd., an independent consulting firm located in Alexandria, VA. His activities ranged from advising small businesses on computer selection, teaching courses on software management, and conducting market analysis, to consulting on software development projects for both defense and commercial contractors. Prior to his retirement from the U.S. Navy in 1978, he was Assistant (for Software Management) to the Director, Computer Resource Office, in the Headquarters of the Naval Material Command. This office is responsible for the policy, standards, and procedures for all computer hardware and software acquisition, development, and application within the Naval Material Command. He was the Configuration Manager of the CMS-2 and SPL/1 programming languages and the Navy member of the DoD High Order Language Working Group that developed Ada. He was responsible for the development of MIL-STD-1679, "Weapon System Software Development." He had earlier served as designer and Project Manager for the development of software for the various Naval Tactical Data Systems. He is a contributing author to three books and has written more than 30 papers and articles, many of which were presented to national audiences.

# Managing Software Development Projects for Maximum Productivity

NORMAN R. HOWES

22

**Abstract**—In the area of software development, data processing management often focuses more on coding techniques and system architecture than on how to manage the development. In recent years, "structured programming" and "structured analysis" have received more attention than the techniques software managers employ to manage. Moreover, these coding and architectural considerations are often advanced as the key to a smooth running, well managed project.

This paper documents a philosophy for software development and the tools used to support it. Those management techniques deal with quantifying such abstract terms as "productivity," "performance," and "progress," and with measuring these quantities and applying management controls to maximize them. The paper also documents the application of these techniques on a major software development effort.

**Index Terms**—Performance evaluation, productivity analysis, progress measurement, software development methodologies, work breakdown structure.

## I. INTRODUCTION

**I**N 1977 we began developing a Project Management System to support worldwide operations. It was designed to assist with the day to day management of large engineering and construction jobs. As a management information system, it was necessary to interface with the company's financial and materials management systems. The result was that these systems had to be totally redesigned to support this new Project Management System.

The overall effort took about two million man-hours. The development of this system called BRICS (Brown & Root Integrated Control System) was managed using the system being developed but in a manual rather than automated mode. The key management concepts used to control this development effort; namely, performance evaluation, multibudgeting, and forecasting with the "variance" technique were documented in [1].

In order to give a complete self-contained treatment of these concepts, the scope of the paper was limited to these (somewhat technical) topics. This necessitated omitting the discussion of other BRICS capabilities such as productivity evaluation. Moreover the successful management of a large-scale software development project involves more than applying techniques such as these. An experienced software development manager has a fixed idea (philosophy) of how software development should be managed. If such a philosophy is a successful one it will automatically tend to maximize productivity and keep the work progressing as planned.

The purpose of this paper is to document such a philosophy.

Manuscript received January 12, 1983.

The author is with Brown & Root, Inc., P. O. Box 3, Houston, TX 77001.

to show the principle techniques necessary to support it, and to point out some common pitfalls that experience teaches one to avoid.

## II. HOW ONE VIEWS SOFTWARE DEVELOPMENT

Scientists know that the way you "look at" a problem often influences whether you can solve it or not. Similarly, your viewpoint influences your ability to manage a software development project efficiently. The author has found it helpful to think of software development management as consisting of two separate but related parts: project planning and project execution. Both parts have five components. The planning components are:

- subdivision of work
- quantification
- sequencing of work
- budgeting
- scheduling.

Subdivision of work is the decomposition of a job into manageable pieces which will be referred to as "work packages." This is sometimes called "packaging the work." Work packages should consist of one generic type of work, should be of short duration, should be logically related to how the work is to be performed, and it should be possible to assign responsibility for the completion of a given work package to one person.

Normally the subdivision of work is arrived at through a series of decompositions based on how the work will be performed. An example of this process is given in Fig. 1. Here, the job of developing the BRICS system was first decomposed into several major components such as developing a requirements specification, translating the requirements into a functional design, expanding the functional design into a technical design, implementing the design (coding), integration testing, acceptance testing, etc.

This is the first level of decomposition. Level 1 components are further subdivided as shown in Fig. 1 to product components at level 2. For example, the technical design component is subdivided into detailed design of processing modules, detailed design of data management modules, calculation of system timings, etc. These level 2 components are further decomposed into level 3 components and so on until the total effort has been subdivided into manageable components (work packages). When the work has been divided in this fashion, the resultant hierarchy is called a "work breakdown structure" (WBS).

Once the work has been subdivided, it can be quantified. Quantification is that component of planning which deter-



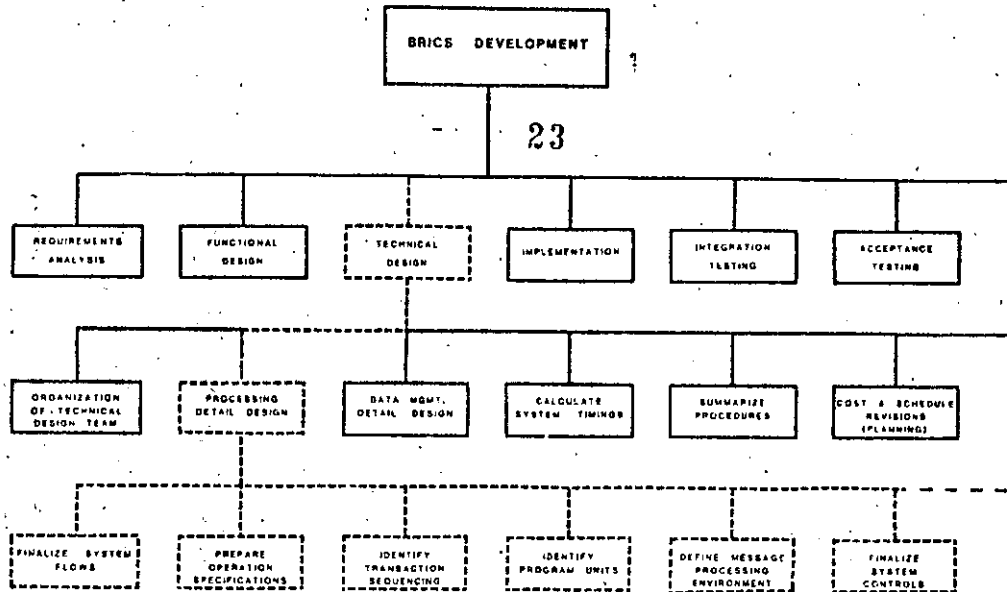


Fig. 1. System development work breakdown structure (WBS).

mines the amount of work (man-hours), overhead, and computer resources required for each work package in the work breakdown structure. The estimated cost for each work package is based on this information.

Fig. 2 shows a copy of the "detailed estimate worksheet" used for quantifying and budgeting work packages for the BRICS software development effort. Note that BRICS work packages were referred to as "control packages" (a departure from standard WBS terminology peculiar to this project). The first step in quantification is to list the activities in each work package and their "unit of measure." The example shown in Fig. 2 is for the FINALIZE SYSTEM FLOWS work package shown in Fig. 1. The unit of measure for activities 2.2.20.1 and 2.2.20.2 is flowcharts. The unit of measure for activities 2.2.20.3 and 2.2.20.4 is reports.

The next step in quantification is to assign quantities to each activity (in the activity's unit of measure). How this was done for work package 2.2.20 (FINALIZE SYSTEM FLOWS) is shown in Fig. 2. The final step in the quantification process is to record the number of man-hours necessary to accomplish each activity in each work package. Depending on who you talk to, the man-hours may be considered as part of the quantification or part of the estimate. For the purposes of this paper it will be considered part of the quantification.

After the work packages have been quantified, the sequence in which the work packages are to be executed needs to be determined. This sequence of work provides the software development manager with an understanding of the order in which the work is to proceed. As the sequence of work is developed, the work breakdown structure needs to be reviewed to ensure that the subdivision of work is compatible with the sequence in which the work is to be completed. This may lead

DETAILED ESTIMATE WORKSHEET						
PROJECT BRICS			PAGE 1 OF 1			
PREPARED BY...		CONTROL PACKAGE 2.2.20		DATE 9-19-79		
ACTIVITY	DESCRIPTION	UNITS	QTY	MHRS	\$	HRF
2.2.20.1	Prepare Final Batch Flowcharts	Charts	60	720		101
2.2.20.2	Prepare Final On-Line Flowcharts	Charts	30	360		102
2.2.20.3	Prepare Batch Flow Narratives	Reports	60	360		
2.2.20.4	Prepare On-Line Flow Narratives	Reports	30	240		
Total				1680		

Fig. 2. Detailed estimate worksheet.

to changes in the work breakdown structure such as changing or creating new work packages to better define the manner and order in which the work is to be accomplished.

After the work packages have been quantified and sequenced, they must be estimated. The estimate for each work package will specify the planned cost to complete the work in the work package. After the estimates are approved by manage-

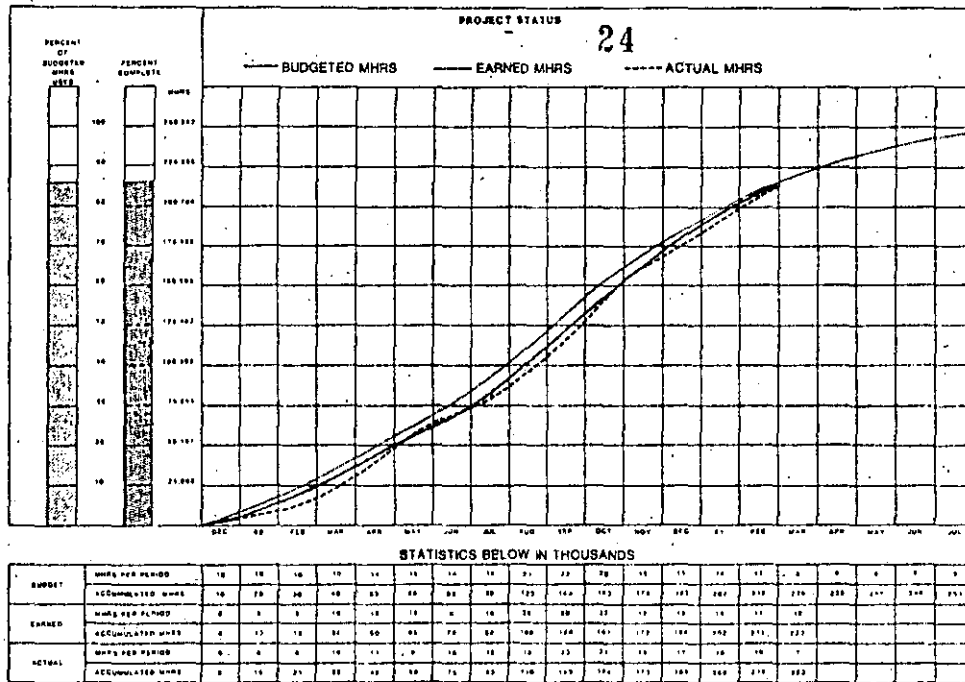


Fig. 3. Project status summary.

ment they will be called "budgets." The man-hours for a given work package are the "man-hour budget" for the package. The terms "work package budget" and "work package man-hour budget" imply a management allocation of resources in terms of cost and man-hours to complete the work package.

Finally, the work packages need to be scheduled to complete the planning process. The purpose of scheduling is not only to predict when a job can be completed given the sequence of work and the resources available but also to establish start and end dates for each work package. The software manager uses these scheduled dates for the work packages to control the work and communicate progress of the work.

III. THE PROJECT PLAN

The budget for your software development effort is the composite of all the budgets for all the work packages in your WBS and the schedule for your project is the composite of all work package schedules. Together the budget and the schedule are referred to as the "plan" or the "baseline."

Again, it is important how this plan is visualized. It is helpful to see an integrated picture of the budget and the schedule components of the plan. This is achieved by using the schedule to "time-phase" the budget. Time phasing shows graphically how the budget is to be expended over time. Fig. 3 shows the time phasing of the manhour budget for the TECHNICAL DESIGN and IMPLEMENTATION branches of the BRICS WBS hierarchy shown in Fig. 1.

The algorithm for time phasing the budget with the schedule is documented in [1]. It can be done manually, but if the

number of work packages in your WBS exceeds 100 it becomes fairly difficult. One of the things BRICS does is to automatically produce time phasing graphs for work breakdown structures of any size whose budget and schedule have been entered into the system.

IV. PROJECT EXECUTION

Basically, the software manager's job is to control the development effort in accordance with the project plan discussed above. The five components of project control are:

- accumulation of actual expenditures
- progress measurement
- performance evaluation
- productivity measurement
- change control and forecasting.

The activities of project control allow the software manager to monitor progress; anticipate and rectify problems; and to continue the "communication" established by the plan to meet requirements, cost objectives and the schedule.

Classical cost accounting methods are used to accumulate actual expenditures of manhours and costs. Each work package is considered as a ledger account and each expenditure incurred for each work package is posted to the appropriate account as it is incurred. Man-hour expenditures should be posted weekly to accommodate the productivity reporting discussed in a later section. Costs can be posted weekly or monthly. For plotting purposes it is advisable to maintain

a historical record of the actual expenditure of man-hours for each work package at the end of each reporting period.

Collecting expenditures at the work package level allows for computing the actual expenditure for any element in the WBS at any level simply by summing. A plot of the actual expenditures against the baseline during the technical design and implementation of BRICS is also given in Fig. 3.

Progress measurement is that element of control that is involved with periodically (usually weekly) determining the status of each work package. Status is measured in percent complete. Usually the best method for measuring percent complete is to compare the actual number of units completed for each activity in a work package with the "budgeted quantity" (quantity shown on the detailed estimate worksheet) for that activity. The ratio obtained is the percent complete for the activity. Percent complete for the work package is computed using the formula

$$\text{WP \% comp} = \frac{\sum (\text{activity \% complete}) (\text{activity man-hour budget})}{\text{work package man-hour budget}}$$

where the summation is taken over all activities in the work package. It is the responsibility of the software manager to insure this data is collected periodically as it is the basis for the calculations used in performance evaluation.

#### V. PERFORMANCE EVALUATION

Performance evaluation is that element of control which compares actual progress and expenditures to the project plan, identifies deviations from the plan and determines solutions to correct for these deviations. Actual expenditures and the baseline are expressed in terms of manhours spread over time as shown in Fig. 3. But progress is measured in percent complete. In order to measure progress in the same units as the budget and expenditures so a comparison can be made one uses the concept of "earned value" (earned man-hours). Earned value (EV) for a work package is defined as

$$\text{EV} = (\text{work package man-hour budget}) (\text{work package \% complete}).$$

Conceptually, earned value represents the (man-hour) value of work accomplished relative to the (manhour) budget. By computing earned value at the work package level it can be obtained for any element in the WBS hierarchy by summing the earned value for all work packages under the given hierarchy element.

A plot of earned value and actual manhour expenditure against the baseline for the BRICS technical design and implementation is given in Fig. 3. This plot is the software manager's principle performance evaluation tool. A detailed discussion of how to interpret such an earned value graph is given in [1]. Basically, if the earned value "curve" is tracking the baseline curve closely, work is progressing as planned and if it is tracking the actual expenditure curve closely, productivity is as planned.

#### VI. PRODUCTIVITY MEASUREMENT

If the earned value curve deviates significantly from either the baseline or the actual man-hour curve, or both there is

reason for concern. It is the responsibility of the software manager to take steps to rectify the problem but before this can be done the problem must first be isolated.

Suppose the earned value curve is tracking the baseline closely but deviates sharply from the actual expenditure curve with the "actuals" curve running "above" the earned value curve. This means the work content is being executed as planned (as scheduled) but the cost in man-hours is significantly more than planned. The obvious conclusion is low productivity. In order to know which work packages are experiencing low productivity, the software manager needs a weekly productivity report.

The calculation of productivity for a work package is as follows: first, the work package is assigned a unit of measure just like the activities in the work package. Each activity in the package may have a different unit of measure and it is not necessary that the work package unit of measure be the same as any of its activities. For instance, the unit of measure for work package 2.2.20 shown in Fig. 2 could be documents and its quantity could be 180.

The work package man-hour budget is obtained by summing the man-hours for each activity in the package. In this case it is 1680 man-hours. Dividing 1680 man-hours by 180 units gives 9.33 man-hours per unit or man-hours per document. This ratio will be referred to as the "budgeted cost per unit" or simply the "budgeted unit rate." Productivity is defined as output per-man-hour. Consequently, the unit rate is the reciprocal of productivity since it is measured in terms of man-hours per unit (output).

The "actual unit rate" for an activity in the work package could be determined by dividing the actual manhour expenditure for the activity by the actual number of units completed on the activity. But to collect costs and compute unit rates for each activity would lead to far too much detail. This is the reason for "packaging" the work in the first place so it can be treated as manageable pieces instead of a mass of detail. What is desired is an actual unit rate for the work package. It is obtained using the formula

$$\begin{aligned} \text{work package actual unit rate} \\ &= \frac{\text{work package actual man-hour expenditure}}{(\text{work pkg. \% complete}) (\text{work pkg. quantity})} \end{aligned}$$

where the formula for work package % complete was given in Section IV. Similarly, budgeted unit rates and actual unit rates can be computed at summary levels of your work breakdown structure by assigning a quantity and unit of measure to each summary level WBS element and using the formulas

$$\begin{aligned} \text{WBS element budgeted unit rate} \\ &= \frac{\text{WBS element man-hour budget}}{\text{WBS element quantity}} \\ \text{WBS element actual unit rate} \\ &= \frac{\text{WBS element actual man-hour expenditure}}{(\text{WBS element \% comp.}) (\text{WBS element quantity})} \end{aligned}$$

where the WBS element man-hour budget and the WBS

PRODUCTIVITY REPORT  
AS OF 06/30/80

26

WBS ELEM	DESCRIPTION	DOC	QUANTITIES			MANHOURS			MANHOURS PER UNIT				
			BUDGET	ACTUAL	COMP. FORECAST	BUDGET	ACTUAL	RED. FORECAST	BUDGET	ACTUAL	FORECAST		
TECH DESG 2.2	TECHNICAL DESIGN	DOC	4214	130	2	4214	92210	1662	2	92210	15.00	12.04	15.00
PRG DESG	PROCESS, DETAIL DESG	DOC	1688	130	4	1688	75151	1662	7	7515	14.90	12.04	14.90
2.2.20	FINALIZE SYST FLOWS	DOC	180	138		178	1680	1662	99	1766	9.33	12.04	11.04
2.2.20.1	BATCH FLOWCHARTS	CHT	40	58	100	58	720	752	100	752	12.00	12.97	17.97
2.2.20.2	ON-LINE FLOWCHARTS	CHT	30	31	100	58	360	480	100	480	12.00	15.48	15.48
2.2.20.3	BATCH NARRATIVES	REP	40	29	50	58	360	732	64	444	6.00	8.60	8.60
2.2.20.4	ON-LINE NARRATIVES	REP	30	20	67	31	240	198	63	294	8.00	9.50	9.50
2.2.22	OPERATION SPECS.	DOC	22	0	0	33	678	0	0	678	19.63		19.63
2.2.22.1	LIST JOB STREAMS	LSI	16	0	0	16	124	0	0	124	7.75		7.75
2.2.22.2	PRELIM OPER SPECS	SPC	16	0	0	16	384	0	0	384	24.00		24.00
2.2.22.3	REVIEW WITH OPS MGMT	RAW	5	0	0	5	120	0	0	120	24.00		24.00
2.2.24	IDENT TRANS SEQUENCE	DOC	33	0	0	33	552	0	0	552	16.73		16.73
2.2.24.1	TRANS SEQ WORKSHEETS	WS	33	0	0	33	528	0	0	528	16.00		16.00
2.2.24.2	TEAM REVIEW OF WS'S	RAW	1	0	0	1	24	0	0	24	24.00		24.00
2.2.25	IDENTIFY PRG UNITS	DOC	734	0	0	734	11684	0	0	11684	15.92		15.92
2.2.25.1	PREPARE PRG CHARTS	CHT	398	0	0	398	6368	0	0	6368	16.00		16.00
2.2.25.2	PREPARE PGB WORKSHEETS	WS	104	0	0	104	1132	0	0	1132	10.88		10.88
2.2.25.3	PREPARE DLI CALL PGM	PGM	104	0	0	104	1132	0	0	1132	10.88		10.88
2.2.25.4	PREPARE ON-LINE MGR	MGR	128	0	0	128	3052	0	0	3052	23.84		23.84

Fig. 4. Productivity report.

element actual man-hour expenditure are obtained by summing the work package man-hour budgets and actual man-hour expenditures for all the work packages under the given WBS element in the work breakdown structure hierarchy, and where the WBS % complete is given by

$$\text{WBS element \% complete} = \frac{\sum (\text{work pkg. \% comp.}) (\text{work pkg. man-hour budget})}{\text{WBS element man-hour budget}}$$

where the summation is taken over all work packages under the given WBS element in the hierarchy.

An example of such a productivity report is given in Fig. 4. The software manager uses the performance report of Fig. 3 together with the productivity report to spot trends and isolate problem work packages. It is also necessary to consult the project schedule to spot problems. Even though productivity is satisfactory, work packages may not be starting or finishing as planned. Such a case would lead to the earned value curve tracking the actual curve closely but deviating significantly from the baseline curve on the performance report.

There are three reasons why software development work does not progress in accordance with the plan. They are:

- 1) changes in the scope of work,
- 2) quantity deviations, and
- 3) productivity deviations.

Changes in the scope of work are redefinitions of the original

requirement. Their basis can range from a change in the user procedures to a better design alternative. Quantity deviations arise from errors in the quantification process and productivity deviation arise from not accomplishing the work at the planned unit rate.

It is important for the software manager to distinguish among these three types of deviations. If work is not progressing as planned because of low productivity, pressure can be applied to increase productivity. Normally, the visibility given to productivity by this management approach tends to stimulate productivity. Applying pressure when productivity meets or exceeds planned unit rates may be counterproductive. Programmers and analysts need to be rewarded for exceeding planned productivity estimates even though the work is not progressing as planned for other reasons. Failure to do so may well introduce productivity problems where you did not have them before.

VII. CHANGE CONTROL AND FORECASTING

It is also important for the software manager to distinguish among the types of deviations in order to "keep the baseline current." This means providing for an up-to-date account of the scope of work and an audit trail of how the original budget evolved into the current baseline. If the baseline is not kept current, the percent complete and earned value computations will not be correct as will be seen in what follows.

A "variance" will denote the documentation of a deviation from the baseline. A "change order" is a variance that rep-

resents an agreed upon change in the scope of work. If the development is being done for a client, a change order will be a client approved variance and may result in a change to the contract. Then the original contract together with all change orders represents the current contractual environment under which the work is performed. The original budget for a work package together with all change orders affecting the package is the "client budget" (called control budget in [1]) for the package. The client budget for the project is the sum of the client budgets for all work packages.

Variances other than change orders will be designated as quantity or productivity variances depending on whether they arose as the result of a (current or projected) quantity or productivity deviation. Sometimes an observed deviation will have both a quantity and productivity component. It is important that the distinction be made and a separate variance be used to document each component. This is because quantity variances will be used to update the baseline whereas productivity variances will only update the forecast.

The client budget for a work package together with all quantity variances that affect the package is the "control budget" (called target budget in [1]) for the package. The control budget for the project is the sum of the control budgets for all work packages. It represents the real scope of work as currently understood and consequently is the true baseline to measure progress against. This is the budget the software manager uses to control the work and consequently this the budget used for calculating earned value.

The control budget for a work package together with all the productivity variances affecting the package is the "forecast" for the package. By constructing the forecast from the budget in this way the difference between the forecast and the budget is automatically quantified and estimated since each variance must be quantified and estimated. A comparison of the budgets and the forecast for a hypothetical development effort is given in Fig. 5.

One of the functional capabilities of BRICS is to provide the user with a means of storing his original quantifications and budgets in the computer and then as time progresses to enter expenditures as they are incurred, progress (percent complete) as it is measured and variances as they are recognized and quantified. BRICS then automatically produces plots of the earned value against the baseline and the "actuals," productivity reports, and an audit trail of how the budgets and the forecast evolved.

As a result the forecast should have more credibility than many of the "subjective guess" forecasts that occur on many software projects. Moreover, the contributions from scope changes, quantification errors, and productivity deviations can be determined. It is impractical to attempt tracking every single deviation from the plan. In practice one relies on the "Law of Compensating Error" to balance out small or insignificant variances and concentrates on tracking the significant ones.

Which variances to track is a matter of judgment, but normally enough of them should be tracked to ensure the forecast is accurate to a tolerance of approximately 5 percent. Also, it is more important to track quantity variances than productivity variances as they affect the baseline. All change orders should be tracked.

## 27 VIII. SELECTING AN APPROPRIATE WBS

There are several software development methodologies on the market. Most of them are in essence a work breakdown structure for software development even though they may not be presented in that format. In any event, they are at least a subdivision of work in that they divide the software development process up into tasks that can be assigned to the analysts and programmers.

Which methodology to choose is probably less important than having a proven methodology and recognizing it for what it is. Many software development projects use these methodologies but few of them use them as the basis for deriving a project plan as described above. It is important that the tasks in the methodology either become the work packages in your subdivision of work or that they be packaged together to form work packages.

These work packages then need to be quantified, sequenced, budgeted, and scheduled as described above. The project plan (baseline) is then produced from this data. During execution of the plan, expenditures need to be accumulated, progress needs to be measured, and variances need to be posted against these work packages. Furthermore, a manual or automated system is needed to produce performance evaluation and productivity reports from these data.

The author has a software development WBS that will be discussed briefly in the next section. However, the BRICS development effort was managed using another methodology. The reader may be interested in the experience and it may shed some light on the problem of selecting an appropriate WBS for your job.

In 1978 Brown & Root purchased a software development methodology called SPECTRUM which is marketed by J. Toellner & Associates. SPECTRUM was used on a small to medium sized application prior to beginning the functional design of BRICS. From what the author could learn from some of those associated with the project, SPECTRUM worked as it was supposed to but the effort required to complete all the SPECTRUM forms was greater than the development effort itself. It may be that this methodology is targeted at larger development efforts and the "overhead" was too great for a smaller project.

So as not to lose their investment but in order to have a more streamlined methodology, Brown & Root undertook to rewrite SPECTRUM. At this time Brown & Root was employing a number of Arthur Andersen consultants and that firm had their own methodology. Both Arthur Andersen and Brown & Root personnel participated in the rewrite. The result was a mixture of SPECTRUM, Arthur Andersen's methodology and Brown & Root experience. This new methodology became the standard for use on the BRICS project. It was now small enough to fit in two rather large ring binders and became known as the "Black Book" methodology (a name derived from its black binders). Since that time Brown & Root has written a much smaller methodology called PROMPT for small to medium sized software projects. The WBS shown in Fig. 1 was extracted from the Black Book Methodology.

The author's experience with the Black Book methodology was that it was still too cumbersome. Many of the tasks were unnecessary and most of them required too elaborate forms that were never used again. The extensive documentation

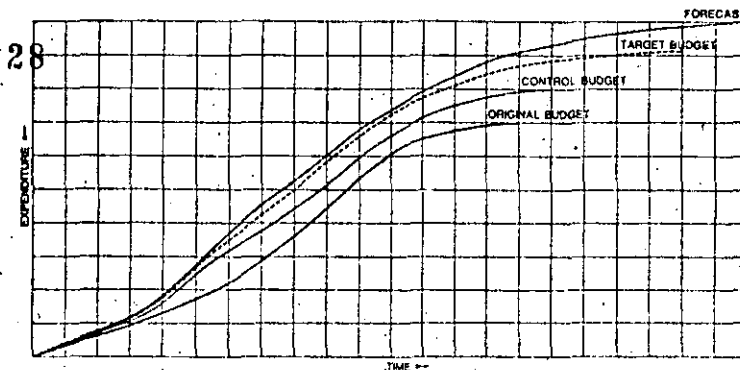


Fig. 5. Budget and forecast comparison.

tended to mask the reason for the individual tasks which often resulted in mechanical completion of forms in order to get a task over with rather than designing the system.

We partially circumvented the problem by distributing the author's methodology to development team members. When tasks were encountered that did not relate to the system being developed, they were "interpreted" in terms of the author's methodology and in several cases management granted permission to substitute other documents for the forms in the Black Book methodology. Proceeding in this manner we managed to finish the technical design phase exactly on schedule and 8 percent under the original estimate. Previously the functional design phase had exceeded the original estimate by 12 percent. In total the project succeeded in completing the design work at a tiny margin under the original estimate and on schedule. Even after acceptance testing the project was less than 6 weeks behind schedule after 4 years development.

#### IX. A PROPOSED METHODOLOGY

The author's methodology is a simple one. The document distributed to team members was only 20 pages. In the face of current thinking in the software engineering field it may seem old fashioned. It is based on the fundamentals of "top-down" architecture but pays little attention to structured programming or some of the activities referred to as structured analysis.

In summary the methodology works like this. First you determine what the system is to do. This is normally documented in something called a requirements specification. It can range from a list of report formats to be produced to satisfy a business application to a formal analysis of how the system is to behave in a real-time environment as, for instance, in an air-defense system. How such a specification was developed for a military command and control system was documented in [2].

From here on the methodology centers around constructing something called a "system flowchart." A system flowchart is no ordinary flowchart like one used to describe a program. The system flowchart is constructed in a series of "levels." In fact, it is not a single chart but a family of charts.

The level 1 system flowchart is constrained to have no more than 6 "boxes" not counting the symbols for inputs, screens, files, and reports. The figure of 6 may seem arbitrary and is.

Another number could be used without altering the structure, but experience has shown 6 to be a good number. The boxes represent processing of some sort. At the first level the boxes usually represent subsystems. The level 1 system flowchart is a "first-cut" at visualizing how the system will be organized at the highest level.

At this point 6 catalogs are begun. These are the component catalog, the input catalog, the report catalog, the screen catalog, the file catalog, and the interface catalog. If a database management system is being used the file catalog may be named something more appropriate like a segment catalog. Each symbol on the level 1 flowchart is assigned an identifier. If the symbol represents a screen the identifier is logged in the screen catalog; if it represents a file it is logged in the file catalog, etc.

Even though the component catalog is limited to 6 components at level 1, the other catalogs are not. As many files, reports, etc. that can be defined at this level should be. The intent is that the level 1 system flowchart should be logically complete and as many files, screens, etc. as are needed to accomplish this is permissible. Also, at this time every member of each catalog must be documented as clearly as possible at this level of detail.

Finally, one normally begins drawing a system hierarchy showing how the components are decomposed at this point. The hierarchy is a shorthand notation for the system flowchart and is valuable for communicating system concepts where the detail of the system flowchart is not necessary.

Next, the level 1 system flowchart is expanded to level 2. Each level 1 component is decomposed into no more than 6 level 2 components. The flowchart is redrawn to reflect the new interfaces among the various level 2 components and new files to accommodate these interfaces, to handle temporary storage, etc. As the flowchart expands, each new interface, each new file and each new component needs to be labeled and cataloged. Just as with level 1, each item in each catalog needs to be documented as clearly and completely as is possible at this level of detail.

This decomposition process continues level by level until you reach components that are too small to decompose further. A general rule of thumb is if a component can be coded with no more than 200 lines of executable code (200 line of code in the procedure division for Cobol programs) that it is unnecessary to decompose it further. These low level compo-

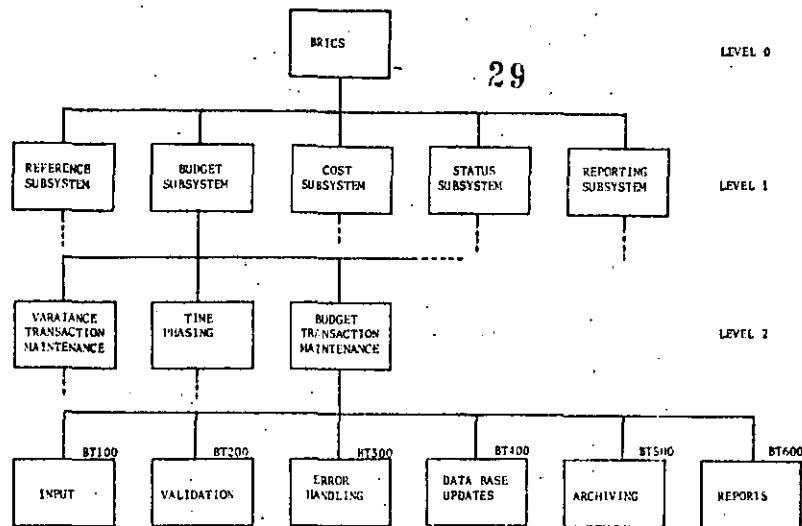


Fig. 6. System hierarchy for BRICS Project.

nents are called modules and eventually become the programs in your system. Figs. 6 and 7 show a simplified level 3 system hierarchy and flowchart, respectively.

When there is no longer anything left to decompose you are through with the design. Detail program specifications remain to be written for each module. Whether these modules adhere to the principles of structured programming or not is of less consequence than the structure induced on the system by this decomposition process. Care should be taken to document each program thoroughly by the liberal use of comment statements.

Modifications to this outline of a methodology will have to be made to accommodate special system requirements such as security, real-time operation, large database requirements, etc. This can be done by adding appropriate work packages to your work breakdown structure.

The Black Book methodology WBS shown in outline in Fig. 1 called for level 1 elements titled functional design, technical design, implementation, etc. Functional design corresponded roughly to developing the system flowchart down to level 2 in the author's methodology. At this point one of the hardest tasks is definition of the interfaces especially when interfacing with systems whose development or maintenance is outside your realm of responsibility.

Technical design corresponded roughly to developing the system flowchart down to level 4, but the parallel was imperfect. The Black Book methodology did not provide for the maintenance of the catalogs of the author's methodology and permitted substituting the system hierarchy diagrams for the system flowcharts. Moreover it possessed many forms to be filled out that were not relevant to the author's methodology.

The BRICS development was accomplished by adding those work packages from the author's methodology needed to develop the system flowchart to the Black Book methodology. The result was satisfactory. Other systems interfacing with BRICS and being developed concurrently used only the Black Book methodology and used only the system hierarchy dia-

grams. These projects had difficulty with the methodology and eventually abandoned it.

#### X. PITFALLS TO AVOID

The advice given in this section is likely to be at odds with the advice you may receive from other quarters. It is only the author's opinion and the only thing the author has to recommend it is that it has worked for the author.

First, do not embark on a large software development project without a proven methodology. It is more important to have a methodology and stick with it than not to use a methodology because of a perceived shortcoming with it. Do not be afraid to modify a methodology to meet your individual requirements. No methodology is universal. A methodology may work well in the hands of its author but not make sense to you. In this regard use common sense. Do not try to use something you do not understand.

Use the methodology to build a work breakdown structure. Base your estimates and schedule on this WBS. Use these to produce a project plan (baseline) and measure performance against it as was discussed in previous sections.

Avoid methodologies that avoid flowcharting. People with nontechnical backgrounds tend to have difficulty with flowcharting and consequently there has been a trend toward replacing them with various hierarchical diagramming schemes. Hierarchical diagrams appeal to our logical intuition whereas flowcharts appeal to our geometrical intuition. They give us a means of visualizing what the system is doing. It is important that the analysts designing the system have a highly developed visualization of the system under development just as an architect can visualize the structure he is designing.

Beware of advice from individuals who tell you that software development is intrinsically different from the development of "tangible" products and as such cannot be quantified and estimated accurately. This usually means they have little experience in the tasks to be estimated. Quantification and estimating may not be easy but they are "do-able."

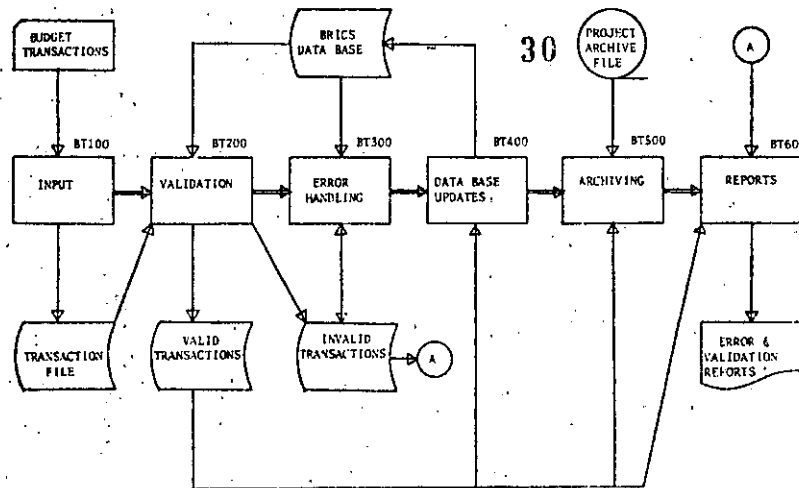


Fig. 7. Level 3 system flowchart for budget transaction maintenance subsystem.

It is true that during quantification and estimating some estimating assumptions may need to be documented as explained in [1, p. 247]. Furthermore, an estimating assumption may prove to be inaccurate at a later date causing a variance to be entered against the baseline. The quantity and criticality of these estimating assumptions will determine the amount of "contingency" built into the estimate, and there is no substitute for experience in correlating cost risk with your estimating assumptions. But the need for estimating assumptions is no reason to discard quantification and estimating as unrealistic.

Furthermore, the vast majority of software development is for products similar in nature to products that already exist, but reflecting the individual requirements of a certain organization. Thousands of multimillion dollar financial systems have been developed in the past and thousands more will be developed in the future. Beware of the one who tries to convince you that the system under consideration is uniquely different from anything in existence. The problem is in matching experience to the work at hand and often those responsible for making DP decisions do not have the background to discern whether the proposed software manager has the experience or not.

Management's need for a reasonably accurate assessment of the cost of development of a new product before deciding to undertake development is universally understood. But in the software development field, software managers frequently encourage general management to undertake risks they do not understand by failing to develop estimates based on detailed quantification.

Finally, avoid the temptation to begin coding before the detailed program specifications have been written for all the modules. Once programming gets underway, maintaining complete documentation will become more difficult. It is best to begin the programming effort with an accurate roadmap. Make implementation a summary level WBS element at a high level. This isolates coding work packages from design work packages.

Think of software as hardware. The modules should become as "chips" or IC's in your mind. Modularity is more important than structured programming. Enforce the limit on the number of lines of code in a module. Each module should be independent of other modules. The goal is to be able to change out modules without affecting other modules just as one changes out a character generator chip to produce a different type font on the screen. This is not a perfect analogy because some of the modules of a software system always correspond to the CPU chip of a computer. Nonetheless, this should be your viewpoint and goal.

You can gain a great deal of insight and advice from others about software development by reading the book containing [3].

#### REFERENCES

- [1] N. R. Howes, "Project management systems," *Inform. & Management*, vol. 5, pp. 243-258, Dec. 1982.
- [2] —, "Development of effective command and control systems," *Signal Mag. (J. Armed Forces Commun. Electron. Ass.)* pp. 44-48, Feb. 1977.
- [3] J. I. Schwartz, "Construction of software, problems and practicalities," in *1974 U.S.C. Seminar: Modern Techniques for the Design and Construction of Reliable Software*, Reading, MA: Addison-Wesley, 1975, pp. 15-54.



Norman R. Howes was born in Kansas City in 1939. He graduated from Eastern New Mexico University, Portales, in 1964 and received the Ph.D. degree in mathematics from Texas Christian University, Fort Worth, in 1968.

He is currently Project Manager for the BRICS project within Land Operations Group at Brown & Root, Inc., Houston, TX. He began his professional career at Texas Instruments in 1966 where he was Head of the Optronics Technical Staff and member of the Computer Advisory Board. He has authored several papers in the areas of mathematics, physics, and computer science and was a member of the Faculties of TCU and the University of Dallas between 1968 and 1971. Thereafter, he was Vice President of Alpha Systems, Inc. In 1973 he joined E-Systems, Inc. as Staff Scientist and later became Computer Consultant to Chief of Defense Denmark. He joined Brown and Root, Inc. in 1978.



# Software Quality Assurance

FLETCHER J. BUCKLEY, SENIOR MEMBER, IEEE, AND ROBERT POSTON, SENIOR MEMBER, IEEE

31

**Abstract**—This paper describes the status of software quality assurance as a relatively new and autonomous field. The history of its development from hardware quality assurance programs is discussed, current methods are reviewed, and future directions are indicated.

**Index Terms**—Development of software quality assurance programs, evaluation procedure, quality assurance, software quality criteria.

## I. INTRODUCTION

As the cost of hardware components continues to decrease and the technology expands along Toffler's curves [1], software continues to insinuate itself into almost all aspects of our lives. A few examples are the following.

1) Financial Systems: Not only are our checking accounts automated,<sup>1</sup> but billions of dollars are shipped electronically every day via Electronic Fund Transfer Systems [2].

2) Transportation Systems: The ubiquitous microprocessor is into automobiles, elevators and escalators, while automated trains, for example, the Bay Area Rapid Transit (BART) system in California, have been providing computerized railway service for some time.<sup>2</sup>

3) Hotel Reservation Systems, Medical Intensive Care Wards, Glass Factories, and Electronic Switching Systems: These have all been automated, and the use of the CAD/CAM processes which develop them is increasing.

Software now permeates the very fabric of our lives. Unfortunately, not only does the software itself continue to fail,<sup>3</sup> but its practitioners consistently fail to meet cost, schedule or technical performance requirements. Yet, despite all the concerns, the promise of software is so bright that more and more is being done using it. Further, due to the need, the number of practitioners is increasing. So in all of this, the question arises, "What is to be done?"

In every time of crisis, every time of troubles, prophets have come roaring out of the desert, preaching baptism and the repentance of sins. The software case is no different. Even today, various groups are taken to the high places and told that salvation is found only in the use of a new higher order language, structured programming, software tools, requirements specification languages, proof of correctness, etc. And the desert is littered with the bones of those who believed and

followed [3]. Recently, a new field has arisen, Software Quality Assurance,<sup>4</sup> which promises much.

The objective of this paper, then is to provide an overview of the emerging field of Software Quality Assurance. This includes:

- 1) the background from which the field emerged;
- 2) a set of definitions to place the field in context;
- 3) a look at the rationale for software quality assurance;
- 4) the organizational implications of software quality assurance;
- 5) current implementations;
- 6) future directions.

## II. BACKGROUND

One acceptable thrust of problem-solving behavior is to look for analogies from another field, and to apply those solutions [4]. Hardware has had Quality Assurance/Quality Control for quite some time. As the value of this field became recognized, its applications to hardware increased and multiplied. These applications have included applications of statistical quality control to inspection of incoming parts, implementation of inspection stations throughout a manufacturing floor, and further on in to manufacturing methods.

The evolution and maturing of the hardware Quality Assurance/Quality Control field can also be seen from the following indicators.

1) A professional society exists to focus its members' interests in the field [5].

2) A series of standards exist throughout the Government and the voluntary standards-making organizations which reflect current practices [6].

Recognizing the value of this existing effort, significant attempts have been made to borrow this QA approach from the hardware field and apply it to software. In the standards area, this borrowing is directly traceable.<sup>5</sup> During this transfer, however, a certain amount of confusion has been incurred, which has led to the need to define more precisely what the field is and the reasons for applying it.

## III. DEFINITIONS

You can wander into any bar in town and get into a fight over quality assurance. To avoid that, and to lay a basis for the rest of the article, the following definitions are presented.

1) *Quality Assurance*: A planned and systematic pattern of

<sup>4</sup> The term "discipline" is sometimes used rather than "field." Recognizing that the term "discipline" brings the image of jackboots crashing down on the pavement, the term "field" is considered preferable.

<sup>5</sup> Military Standards include MIL-S-52779A. The FAA has FAA-STD-018, while in the voluntary standards area, ANSI/IEEE 730-1981, *IEEE Standard for Software Quality Assurance Plans* has been approved.

Manuscript received November 15, 1982; revised November 1, 1983.

F. J. Buckley is with RCA, Moorestown, NJ 08057.

R. Poston is with Programming Environments, Inc., Way Side, NJ 07764.

<sup>1</sup> One method of checking how good a bank's EDP system is, is to see where the bank's programmers keep their money.

<sup>2</sup> As one wag put it, "As computers continue to intrude into our transportation networks, jogging will become a more popular sport."

<sup>3</sup> None of these failures are trivial. Consider the case of a large hotel in Chicago whose reservation system failed at 3:00 PM on a busy day. With no backup, the management had to go from room to room to physically ascertain which rooms were occupied.

all actions necessary to provide adequate confidence that the item or project conforms to established technical requirements<sup>6</sup> [7].

2) *Quality*<sup>7</sup>: The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs [8].

3) *Quality Control*<sup>8</sup>: Those Quality Assurance actions that provide a means to control and measure the characteristics of an item, process or facility to established requirements [9].

It is perhaps at the term "quality control" that the hardware analogy begins to fail. The term appears to arise principally in the fabrication stages of hardware manufacture, particularly in those processes which the same type of object is repeatedly produced. This view usually does not carry over to software in which the coding is really a continuation of the design effort and only one part is produced.<sup>9</sup>

#### IV. RATIONALE FOR SQA

Given the above, the question then arises, why are we concerned with software quality assurance? In reality, there appears to be three main reasons.

1) *Legal Liability*: When catastrophic strikes, the normal red-blooded American reaction is to sue.<sup>10</sup> When the case gets before a judge, one of the determinations to be made is whether or not the developer of the software acted as a reasonable and prudent person should have acted in the development of software. The follow-on is to determine if there is a

<sup>6</sup> There are two other accepted definitions of Quality Assurance:  
a) "All those planned or systematic actions necessary to provide adequate confidence that a product or service will satisfy given needs" (ANSI/ASQC A3-1978).

b) "All those planned and systematic actions necessary to provide adequate confidence that an item or a facility will perform satisfactorily in service" (ANSI N45.2-10-1973).

<sup>7</sup> In the software world, a metric definition of the term "quality" has been projected as follows:

"Quality: The degree to which software conforms to quality criteria. Quality criteria include but are not limited to:

- |                     |                 |                   |
|---------------------|-----------------|-------------------|
| • Economy           | • Correctness   | • Resilience      |
| • Integrity         | • Reliability   | • Usability       |
| • Documentation     | • Modifiability | • Clarity         |
| • Understandability | • Validity      | • Maintainability |
| • Flexibility       | • Generality    | • Portability     |
| • Interoperability  | • Testability   | • Efficiency      |
| • Modularity        | • Reusability   |                   |

(These may also have subgroups)"

There are two difficulties associated with the metric definition of quality:

a) To be of value, the metric definition must be defined so that its achievement can be determined in an objective manner. Otherwise, the determination and the summation become a subjective judgment in which goodness is in the eye of the beholder.

b) The metric definition may have no value in a particular project. For example, the software will be of higher quality if it is transportable between an Intel 8080 and a Cyber 175, but who cares? (Even more important, who wants to spend real money pursuing an ethereal goal of higher quality software when the pursuit makes no sense?)

<sup>8</sup> An alternative definition is as follows:

*Quality Control*: The operational techniques and the activities that sustain a quality of product or service that will satisfy given needs; also the use of such techniques and activities (ANSI/ASQC A3-1978).

<sup>9</sup> There is, however, a viewpoint that states that we do have a fabrication stage in the software life cycle and that it takes place at the point where we are stating how good the design is.

<sup>10</sup> There is a preliminary step as pointed out by a lawyer in commentary on an earlier draft; that step is to determine that the person to be sued has money.

standard established by a representative consensus of industry professionals [10]. This, then, was the basis for the development of ANSI/IEEE 730-1981.

2) *Cost Effectiveness*: Software is expensive, and there appears to be significant savings achieved by good SQA.<sup>11</sup> In this case, the form of the organization and the manner in which the cost codes are structured can significantly affect both the viewpoint and the implementation.<sup>12</sup>

3) *Customer Requirements*: Many customers have been burned badly by software developers and now require certain SQA actions or even a SQA program. In this case, the actions are externally driven.<sup>13</sup>

Given the rationale, then, how do we organize and what do we do?

*Organizational Implications*: Throughout the hardware world there is a traditional view that QA/QC is an organization. Looking at this historically, this is one of the strong roles that has been played. As the product flowed through Engineering, Purchasing, Drafting, Manufacturing, and other independent entities, it was the QA/QC department that inspected and judged. However, in software the analogy does not appear to hold. Organizational models in the software world include a project manager who draws resources from a matrix organization. The project manager has total responsibility for the software and thus there is no need for the historical role of SQA as an integrating organization. Despite the failure of that hardware management analogy, SQA is becoming a more popular organizational element. The reasons for this appear to include integration and independence.

1) *Integration*: A good many software project managers are short-term goal oriented. They are judged on a project-by-project basis, with demanding schedules, not enough money, a lack of the skilled manpower to execute the effort, etc. They do not have the ability to amortize upgrading actions, e.g., software tools and training, over more than one project, nor do they have the time to reflect on better ways to develop software.<sup>14</sup> The role of SQA is then:

a) to find the better way, from a long-range viewpoint, over the course of all the software projects in the plant;

b) to educate all those involved in developing the product in the implementation of the better way.

2) *Independence*: The second organizational rationale is that of independence—that there are certain items best not

<sup>11</sup> The most immediate reason is the relative cost of finding errors in test activities as opposed to the cost of finding them earlier; e.g., in design reviews.

<sup>12</sup> Consider the case of a development group which has no responsibility for maintenance. Unless there is some strong driver, there will be little attention paid toward building a product which can be maintained.

<sup>13</sup> These actions, on the customer's part, may in part be self-defeating. Consider the case of a customer who perceives that whatever he does, the software is going to be a troublesome spot. When the contract is over, the customer still wants to survive. One attitude taken is to blindly apply military standards so as to be able to say, when it is all over, "Yes, Boss, I know they got into trouble, but I applied all the standards ahead of time, to try to keep them clean."

<sup>14</sup> One software project manager described himself just like Winnie the Pooh going bump, bump, bump down the stairs. He knew there must be a better way, if only he could stop for a moment and think about it.

done by the software developer. These include:

- a) configuration management of the code;
- b) reviews and audits;
- c) test/verification.

Throughout all of this organizational ferment to date, there has been no consensus on a preferred solution, no approved type of SQA organization. Even further, the consensus is that SQA is a function, not an organization, and that different companies and groups will organize themselves differently under different names to meet their own situations.<sup>15</sup>

A few examples include the following.

- 1) One company that has a separate group, Technical Assurance, which is responsible for the independent technical review of both hardware and software. This group reports directly to the Chief Engineer.
- 2) A second company that includes configuration management of software as an organizational part of their SQA activity.
- 3) A third that includes substantial approval and test functions.

In addition, certain segments of the data processing industry identify a database administrator to accomplish much of what would be otherwise assigned to a Quality Assurance Department in an aerospace-industry organization.

Current approaches to putting Quality Assurance into practice are to minimize role conflict and clarify interfaces [11]. Placed in that perspective, the precise organizational details fade into second-level concerns.

#### V. CURRENT IMPLEMENTATIONS

To judge SQA by the current organizational implementations becomes a snare and a delusion. As pointed out above, SQA is not an organization; it is rather, "A planned and systematic pattern of all actions . . ." Therefore, to judge SQA, those actions, implementations, functions themselves should be assessed with the follow-on question being: What is the minimum subset of SQA actions?

The view, herein, is to utilize the items identified in ANSI/IEEE STD 730-1981 as the consensus items, recognizing that standard had legal liability as its basic rationale. That standard identifies the following items as required:

- 1) management
- 2) documentation
- 3) standards, practices, and conventions
- 4) reviews and audits
- 5) configuration management
- 6) problem reporting and corrective action
- 7) tools, techniques, and methodologies
- 8) code control
- 9) media control
- 10) supplier control
- 11) records collection, maintenance, and retention.

<sup>15</sup> Consider that, as pointed out by R. Peach, Chairperson of the ANSI group that produced ANSI/ASQC Z-1.15-1979, *Generic Guidelines for Quality Systems*, that the Japanese do not have strong QA organizations.

On the issue of how independent the SQA organization should be, consider that Juran recommends for large projects, that they report to the project manager. (*Quality Control Handbook*, J. Juran, 3rd ed., Chapter 44, McGraw-Hill, New York, 1974.)

#### A. Management

From the standpoint of management techniques, we are very well off indeed. Work breakdown structures [12], [13], cost estimating methods [14], and scheduling approaches [15] are common in the state of the art. In certain instances, software projects have been completely successful [16], [17]. However, despite the outpouring of all the past 20 years, the manifestations of massive software management problems<sup>16</sup> continue to be exhibited, i.e., excessive costs, schedule slippages/delays, and excessive errors and faults [18].

Faced with this, the question then arises: What are the major problems of software management and why are we continuing to fail? From a software manager's perspective, there are two overwhelming problems [19]: inadequate planning and inadequate requirements specifications.

1) *Inadequate Planning*: A planning activity normally takes in, as input, an idealized model of the process to be executed. Adaptations are then made based on the project to be executed, the customer's desires, and the experience of the manager. The output of this activity is a series of documents, project management plans, schedules, costs, computer program development plans, etc. However, the idealized model of the process is that of a software life cycle model, which projects that a good requirements specification will exist by the end of a specific phase.

2) *Inadequate Requirements Specifications*: The problem of inadequate requirements specifications is an old problem and is related to two factors.

a) The first is a human factors problem. We have an inability to grasp a totality and a further inability to communicate what we do grasp. This is reflected into specifications that lack completeness, clarity, and consistency.

b) The second is the environmental interaction. When a new realization is evolving, it affects the environment in which the system is to be emplaced. Hence, second-order effects occur and requirements grow. This again affects the completeness of our specifications. Our specifications are becoming better in terms of consistency with the application of software tools, machine-processable languages, etc. The influx of visual arts graduates has materially assisted the efforts aimed at clarity. Alas, however, for completeness, there is no known cure.<sup>17</sup> The impact on this is to invalidate the software life cycle, the idealized model of our process, and thus to provide built-in overruns.

3) *Assessment*: From an overall assessment, we can do well those things we have already done, e.g., Airline Reservations Systems. But, fortunately, or unfortunately, the applications

<sup>16</sup> The General Accounting Office Report FGMSD-80-4, "Contracting for Computer Software Development," 9 September 1979 stated:

- 1) 50% + of contracts had cost overruns
- 2) 60% + of contracts had schedule overruns
- 3) 45% + of software contracted for could not be used
- 4) 29% + of software contracted for was never delivered
- 5) 19% + of software contracted for had to be reworked to be used
- 6) 3% - of software contracted for had to be modified to be used
- 7) 2% - of software contracted for was usable as delivered.

<sup>17</sup> As Frosh said "The idea of a complete specification is an absurdity." (R. A. Frosh, "A new look at systems engineering," *IEEE Spectrum*, Sept. 1969.)

continue to expand in both volume and new fields.<sup>18</sup> To resolve this, the field appears to be evolving towards:

- a) recognizing as a way of life that requirements will continue to be incomplete;
- b) encouraging schedules that explicitly recognize incomplete requirements;
- c) promoting early identification of requirements specification changes—and correct disposition;
- d) ensuring that software is designed to enhance change implementation.<sup>19</sup>

#### B. Documentation

The need for documentation during software development has long been recognized. Standard formats for documentation of requirements and design documentation are plentiful and easily tailored [20]. The difficulty is not with the document formats, the difficulty is to determine what goes into the documentation.

- 1) The problems with software requirements specifications have been covered earlier.
- 2) Extending the requirements into the design has been a matter of attack for many years. A number of design strategies exist [21]. From a QA standpoint, which one is used is not important. It is important that one be chosen and implemented. Today, this still does not appear to be a widespread practice.
- 3) Verification and validation, and testing<sup>20</sup> are becoming more of a science and less of an art. Despite the emphasis of collecting error data [22], and the efforts of many to define the field [23] and act on it [24], this collection of software activities remains relatively uncivilized [25], [26]. The forces appear to be split between the academic thrusts towards "proof of correctness" and the industrial mucking through the mire.<sup>21</sup>

<sup>18</sup> Software appears to be continuously driven, as Kirk said, "... to boldly go where no man has gone before ..." (J. T. Kirk, *Voyages of the Starship, Enterprise*, Stardate 8206:04.)

<sup>19</sup> This is the thrust towards independent modules, in terms of module coupling and module strength. See, for example, G. Myers, *Composite/Structured Design*. New York: Van Nostrand Reinhold, 1978.

<sup>20</sup> The words are used in the following context.

1) *Verification*: The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.

2) *Validation*: The process of evaluating software at the end of the software development process to ensure compliance with software requirements.

3) *Testing*: The process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

These definitions are in accordance with ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology.

<sup>21</sup> For one view of the usefulness of "proof of correctness," see P. Moranda, *op. cit.* There has been a strong thrust towards the realization of formal verification techniques by the Department of Defense since 1978. This effort has been aimed at the computer security issue, particularly at the multicompartimented secure operating systems area. See, for example,

a) J. D. Tangney, *History of Protection in Computer Systems*, Mitre Tech. Rep. 3999, Mitre Corp., Bedford, MA, July 15, 1980.

b) J. M. Silverman, *Proving an Operating System Kernel Secure*, 815RC31, Honeywell Syst. and Res. Cent., Minneapolis, MN, Apr. 1981.

The application of this technology to other, non-DoD efforts, e.g., Electronic Funds Transfer, is obviously desirable and could yield a significant commercial advantage to both ADP vendors and users.

Fully independent efforts using outside contractors are gaining popularity in some portions of the commercial world but enough examples of inadequate verification have been documented in the literature to where what is happening remains an open question.<sup>22</sup>

#### C. Standards, Practices, and Conventions

There is, today, a plethora of standards, practices, and conventions that cover the waterfront, to include as mentioned above, Documentation, Logic Structure, Coding, and Commentary. They have not yet all been collected into volumes and published, as NBS has done for documentation [27], but enough books have been published so that anyone in need can abstract at will.

The major difficulties with the standards have not been with their existence but in their reasonableness and their ability to be implemented.

Standards, practices, and conventions can be imposed by fiat. The effectiveness of that approach versus a consensus approach remains to be measured [25].

The ability of the standard to be implemented depends on two factors:

- 1) the environment in which the standards are embedded;
- 2) the ability to determine objectively if the standards are being followed.

Both of these factors are leaning more in the direction of implementation into an overall programming environment in which the standards, practices, and conventions are human-factored into the software supporting the development team. If adherence to the standards is assisted by the support software, a major step towards effective implementation has been reached.

#### D. Reviews and Audits

The industry is becoming more conscious of the values of reviews and audits and the methodology for accomplishing them.

Design and code inspections, reviews, and audits are becoming common in the state of the art [29]. The difficulty in implementing them is that to date, a comprehensive justification based on errors found at reviews versus errors caught at test time, in an industrial environment does not appear to have been published. Thus each company implementing such actions is usually, doing so on a hesitation basis pending accumulation of sufficient statistics to cost-justify the effort.

<sup>22</sup> For example, NUREG-0653, *Report on Nuclear Industry Quality Assurance Procedures for Safety Analysts Computer Code Development and Use*, U.S. Nuclear Regulatory Commission, Aug. 1980 provides the following examples of industry practices on pp. 18-20.

- 1) Reliance on the individual doing the development to assure the code does not contain errors.
- 2) Reliance on the developer to perform verification.
- 3) Allowing the developer to determine the extent of checkout and verification.
- 4) Poor documentation or no documentation at all.
- 5) Verification requirements being waived.
- 6) Vendors using codes developed by other organizations having no requirements that the codes be developed in accordance with any Quality Assurance Procedures.

The most interesting item was that, despite these examples, the conclusion of the report was that Nuclear Industry Quality Assurance procedures for Safety Analysis Computer Code development and use were basically sound.

### E. Configuration Management

Configuration Management methodology is available, and several tool sets are also commercially available and commonly used [30]. The implementations vary in depth according to the project parameters. This is normally combined with code control, media control, and problem reporting and corrective action as a discipline. There have been, however, sufficient reports from the field that a conclusion can be drawn that the importance of code control is not fully understood.<sup>23</sup>

### F. Tools, Techniques, and Methodologies

Special software tools have been either made by the users or bought for many years. There appear to be today a proliferation of tools both free and for sale [31]. Further, the tools themselves are becoming integrated into complete programming environments [32].

## VI. FUTURE DIRECTIONS

The industry appears to be implementing Software Quality Assurance in a phased approach recognizing it as a series of functions to be tailored to fit particular organizations.

One consultant's report summarized the field as follows [33].

- 1) The current lore of the field is that:
  - a) quality is designed into software, not tested in;
  - b) user participation should be maximized;
  - c) SQA should be involved from the beginning of the project and participate in each stage;
  - d) the earlier errors are found and corrected, the more total life cycle costs can be reduced;
  - e) SQA should be central and independent.
- 2) Concerns associated with SQA include:
  - a) the recognition of the need to control changes to software and to keep documents up to date with the code;
  - b) the danger of over-regulation;
  - c) the fear of empire building associated with the fear that SQA will be a hinderance rather than an aid;
  - d) the difficulties associated with finding personnel who have qualifications necessary for an effective SQA team.
- 3) SQA efforts in the commercial world are proceeding generally in phased implementations, roughly as follows.
  - a) *Minimum Efforts*: These include:
    - i) establishing a library of standards, procedures, and technical publications;
    - ii) extracting/creating a usable set of guidelines;
    - iii) establishing change control procedures;
    - iv) reviewing documents for completeness and conformance.
  - b) *Mid-Level Efforts*: These include the minimum ef-

<sup>23</sup> For example, NUREG-0653, *op. cit.*, points out:

- a) One organization did not procedurally control the follow-up of codes with known errors (p. 25).
- b) Another organization did not procedurally control the follow-up and closeout of problems identified with code (p. 25).
- c) In one case the only way code modifications and/or changes in code status were transmitted to the users was by the code custodian informally telling the users with no formal transmittal identifying the changes (p. 24/25).
- d) In one case it was found that more than one version of a code existed yet both versions had the same identification (p. 24).

35

forts plus:

- i) establishing configuration management procedures;
  - ii) reviewing documents for content, consistency, and quality;
  - iii) acting as a consultant and advisor throughout the life cycle, by participating in developing documents, change control boards, walk-throughs, reviews, and audits.
- c) *High-Level Efforts*: These include the previous items plus:
- i) preparing comprehensive system test plans;
  - ii) conducting independent, in-house system integration tests;
  - iii) preparing test analysis reports;
  - iv) maintaining a baseline library for all project documentation and code;
  - v) maintaining a test library of test plans, test reports, and evaluation of testing techniques;
  - vi) maintaining currency with state-of-the-art analysis, design, programming, and testing technologies.

## VII. CONCLUSIONS

The tasks required of software grow increasingly greater, constantly challenging our abilities to fulfill them. In response to that challenge, a field has been borrowed from hardware, that of Quality Assurance, modifying it to allow for software differences. Recognition of the field and its effective application is rapidly growing.

## REFERENCES

- [1] F. Toffler, *Future Shock*, Bantam Edition, 1971.
- [2] For more detail on this, see the Special Issue on Electronic Funds Transfer, *Commun. Ass. Comput. Mach.*, vol. 22, Dec. 1979.
- [3] For a somewhat pragmatic view of promises, see P. Moranda, "Software quality technology (Sad), status of: (Unapproached). Limits to: (Manifold) alternatives to," *Computer*, pp. 72-78, Nov. 1978.
- [4] G. Polya, *Induction and Analogy in Mathematics*. Princeton, NJ: Princeton Univ. Press, 1954, ch. 2.
- [5] The American Society for Quality Control (ASQC), 161 West Wisconsin Avenue, Milwaukee, WI 53203.
- [6] Government Standards include:
  - a) The Military Standards, e.g., MIL-Q-9858A "Quality Program Requirements," and MIL-I-45208 "Inspection System Requirements" (copies of military standards are available from the Naval Publications and Forms Center: (Code 301), 5801 Tabor Avenue, Philadelphia, PA 19126).
  - b) Federal Aviation Administration Standard: FAA-STD-013.
  - c) The Nuclear Regulatory Commission Requirement: 10 CFR 50 Appendix B.

An index to voluntary standards can be found in the *Catalog of American National Standards*, ANSI, 1430 Broadway, New York, NY 10018.
- [7] ANSI/IEEE Std. 730-1981, *IEEE Standard for Software Quality Assurance Plans*, 345 East 47th Street, New York, NY 10017, Nov. 13, 1981.
- [8] ANSI/ASQC A3-1978.
- [9] ANSI N 45.2, 10-1973.
- [10] For more details on this, see H. Truheart and S. Green, "Professional liability and the use of computers," in *Proc. ASCE 1st Int. Convention of Computers in Civil Eng.*, 1981.
- [11] See G. Tice, "Software quality control—A roadbed for the bullets," *1980 ASQC Western Regional Conf.* for a detailed discussion of the psychological acceptance factors required to implement SQA on a cost-effective basis.
- [12] R. C. Tausworth, "The work breakdown structure in software project management," *J. Syst. Software*, vol. 1, pp. 181-186, 1980, Elsevier/North-Holland.

- 36
- [13] MIL-STD-881A, *Work Breakdown Structures*.
- [14] L. H. Putnam and R. W. Wolverton, *Tutorial Quantitative Management, Software Cost Estimating*. IEEE Comput. Soc., 1977.
- [15] V. G. Reuther, "Using graphic management tools," *J. Syst. Management*, vol. 30, pp. 6-17, Apr. 1979.
- [16] E. B. Daly, "Management of software development," *IEEE Trans. Software Eng.*, pp. 229-242, May 1977.
- [17] P. C. Belford, R. A. Berg, and T. L. Hannan, "Central flow control software development: A case study of the effectiveness of software engineering techniques," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 85-93.
- [18] B. DeRose and H. Nyman, "The software life cycle—A management and technological challenge in the Department of Defense," *IEEE Trans. Software Eng.*, pp. 309-311, July 1978.
- [19] R. Thayer, "Major Issues of Software Engineering Project Management," *IEEE CompSoc*, 1978.
- [20] See, for example, ANSI/IEEE Std 829-1983, IEEE Standard For Software Test Documentation; IEEE Std 830-1983, IEEE Guide for Software Requirements Specifications, Federal Information Processing Standard Publication (FIPS Pub) 38 *Guidelines for Documentation of Computer Programs and Automated Data Systems*, Nat. Bureau of Standards, Feb. 15, 1976, and FIPS Pub 64 *Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase*, Nat. Bureau of Standards, Aug. 1, 1979.
- [21] An overview of the field is provided by Bergland, G. D., "A guided tour of program design methodologies," *Computer*, pp. 13-37 Oct. 1981.
- [22] See, for example:
- J. L. Elshoff, "An analysis of some commercial PL/I programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 113-120, June 1976.
  - E. F. Miller, "Some statistics from the software testing service," *ACM SigSoft. Software Eng. Notes*, vol. 4, pp. 8-11, Jan. 1979.
- [23] For example, M. S. Deutsch, "Software project verification and validation" (Tutorial Series 7), *Computer*, pp. 53-70, Apr. 1981 and NBS Special Publ. 500-75, *op. cit.*
- [24] C. Gannon, "Error detection using path testing and static analysis," *Computer*, pp. 26-31, Aug. 1979.
- [25] R. L. Glass, "Real-time: The 'lost world' of software debugging and testing," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 264-271, May 1980.
- [26] R. House, "Comments on program specification and testing," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 324-331, June 1980.
- [27] FIPS Pub 38, *op. cit.*
- [28] For a thoughtful approach to standards, see D. T. Ross, "Homilies for humble standards," *Commun. Ass. Comput. Mach.*, vol. 19, Nov. 1976.
- [29] For example, see:
- M. G. Fagan, "Design and code inspections to reduce error in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 219-248, 1976; for one implementation.
  - The Ethnotechnical Review Handbook*, Ethnotechnical Press, 1980, P.O. Box 6627, Lincoln, NE 68506 for detailed insight on how to conduct informal reviews.
  - Control Objectives*, EDP Auditors Foundation, 1980, Publications Office 1468, Altamonte Springs, FL 32701 for a comprehensive auditing guide.
- [30] See, for example,
- IEEE Std 828-1983, IEEE Standard for Software Configuration Management plans.
  - Tutorial: Software Configuration Management*, IEEE Comput. Soc., 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.
  - E. Bersoff, *Software Configuration Management*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
  - C. R. Fredrick, "Project implementation of software configuration management," in *1981 ACM Workshop/Symp. Measurement and Evaluation of Software Quality*, pp. 49-56.
- [31] One catalogue of such tools is provided by D. G. Reifer, *Software Tools Directory*, Reifer Consultants, Inc., 2733 Pacific Coast Highway, Suite 203, Torrance, CA 90505.
- [32] For:
- An overview of UNIX, one of the more popular systems, see B. W. Kernighan and J. R. Washey, "The UNIX programming environment," *Computer*, pp. 12-24, Apr. 1981.
  - Near term research directions, see L. Osterweil, "Software environment research: Directions for the next five years," *Computer*, pp. 35-43, Apr. 1981.
  - Examination of costs, benefits, and other issues, together with an analysis of some currently implemented programming environments, see D. Prentice, "An analysis of software development environments," *ACM SigSoft Software Engineering Notes*, vol. 6, pp. 19-27, Oct. 1981.
- [33] Private communication.



Fletcher J. Buckley (S'60-M'62-SM'76) received the engineering degree from the United States Military Academy, West Point, NY, in 1954, and the M.S.E.E. degree from Stanford University, Stanford, CA, in 1961.

In 1975, he joined the Software Development Activity at RCA, Moorestown, NJ. His interests include project management of real-time software systems. He is currently the Chairperson of the Software Engineering Standards Subcommittee of the IEEE Computer Society Technical Committee on Software Engineering.



Robert Poston (M'71-SM'81) is President of Programming Environments, Inc., Way Side, NJ. He has 20 years of experience managing and developing software for process control, simulation, radar, communication, numerical control, business, and operating systems. From 1978 to 1982 he chaired the IEEE Software Engineering Standards Subcommittee. The first IEEE Software Engineering Standards Application Workshop (SESAW) was initiated under his leadership. He has lectured at numerous technical symposia in this country and abroad. In 1983, he was able to present his views about Standards to professionals throughout the People's Republic of China while touring there with the U.S. People-to-People Computer Software Delegation. Currently, he is the Standards Coordinator between the IEEE Computer Society and the American National Standards Committee, X3, Technical Director of IEEE Seminars on Software Engineering Standards, and Standards Editor of IEEE SOFTWARE Magazine.

# The Software Engineering Shortage: A Third Choice

JAMES P. MCGILL 37

**Abstract**—As interest in the concepts and methods of software engineering increases, many companies, particularly in aerospace, find it difficult to acquire software developers with the desired skills. The option of full-time, company-based training is discussed with suggestions for implementation. Lessons learned from the actual implementation of such a program are discussed along with possible directions for future evolution.

**Index Terms**—DSDD, industrial training, software engineering, software life cycle.

SEVERAL books and magazine articles have appeared in recent years chronicling the recognition of a "software crisis" in the late 1960's and subsequent attempts to deal with it [1]. Essentially, the crisis referred to the fact that as large development projects, involving both hardware and software, began to be undertaken, it was discovered that the hardware was often delivered in working condition while the software was either not delivered or required extensive reworking upon delivery. The reaction to this problem was a careful examination of typical software development techniques, which led to the discovery that they were generally chaotic and abetted a more fundamental problem, the inability of humans to communicate effectively with one another. The result has been the emergence of several modern methodologies all designed to overcome the communication problem by stressing requirements analysis, modular design, structured programming, and other procedures that promote a clear understanding of the problem and a disciplined, thoroughly documented approach to a solution.

The gradual shift from haphazard development to the methodical engineering of software has led to the emergence of a new type of engineer, the *Software Engineer* [2]. A software engineer may be defined as an individual who is skilled in the application of sound, established engineering and management principles to the analysis, design, construction, and maintenance of software and its associated documentation. Despite the inescapably logical arguments which can be made for the modular techniques, industry has been slow to incorporate or even recognize the need for them or those who practice them. This situation is now changing [3].

The Department of Defense, too often the recipient of poorly developed software, is becoming aggressive in its support of the newer techniques of software development and management. Those industries, such as aerospace, which interface heavily with the DoD, are becoming very interested in hiring software engineers. The need for these professionals has been recognized.

Manuscript received December 1, 1982.

The author is with the Lockheed Missiles and Space Company, Inc., Department 62-M4, Building 581, P. O. Box 504, Sunnyvale, CA 94086.

## A SHORTAGE OF SOFTWARE ENGINEERS

The services of software engineers who understand the software development cycle and the tools and methods applicable to each phase of that cycle, are being sought nationwide. Unfortunately, the supply of such persons is quite small. Papers have appeared proposing model undergraduate and graduate curricula for degrees in software engineering [4], [5]. A handful of universities have begun to experiment with such curricula. Indeed, Seattle University and the Wang Institute have just recently awarded the nation's first master's degrees in software engineering. But the supply of new graduates benefiting from these programs does not come close to satisfying the current demand. Therefore, a company seeking the services of a software engineer frequently is left with two choices:

1) assuming that no software engineering graduates are available, it can hire someone with a degree in computer science, mathematics, electrical engineering, other technical discipline, and let that person learn the software development life cycle through on-the-job experience, perhaps supplemented by seminars; or

2) the company can attempt to lure established software engineers away from other companies.

Neither of these solutions is satisfactory. A new graduate lacking experience in large-scale software development will require a break-in period which could last months or years before s/he become productive. During this period, the individual's contributions to the effort could actually be detrimental if not strictly controlled.

Robbing Peter to pay Paul is also no solution. It merely shifts the problem to another company. The industry-wide problem remains.

## A THIRD CHOICE

There is a third alternative, however. By establishing a program for cross training some of their own experienced engineers in the disciplines of software engineering, companies could create a continuous source of such talent. The goal of such a program would be to produce software development specialists trained in the recognized foundation areas of computer science, management techniques, communication skills, problem solving, and design theory [6]. A large company undertaking such a program enjoys significant advantages over universities. These include the following.

- Once committed, the company typically has more financial resources to devote to the program.
- The company can reasonably expect selected candidates to be more mature, motivated, and experienced than college students.
- The company typically has experienced software de-

velopment personnel. Such persons can make excellent lecturers on software development topics.

• The company environment provides a unique opportunity for research into the effectiveness of new methods of software development. Feedback from such monitoring can quickly be incorporated into the company training program. Universities would require much longer to react to such feedback.

• The training may be slanted directly toward specific company needs, further reducing the break-in period after graduation from the program.

Ideally, graduates should be immediately useful and productive upon joining their respective development projects. The students selected for the program should, therefore, not be newly hired college graduates (although a modified program for new hires is certainly feasible). They should already have some familiarity with the company. They should have a record of demonstrated competence in computer related technical disciplines (math, physics, programming).

Pursuant to this goal of immediate postgraduate productivity, the training program must expose students to a variety of subjects. Defense contractors, for example, typically find themselves developing software of a highly technical nature. The training program, in such a company, might be obliged to include courses in mathematics, systems engineering, and physics. The desirability, in any company, to complement disciplined, modular program design with similarly disciplined coding techniques implies the need for familiarity with the concepts of structured programming. Competent instruction in Pascal or Ada<sup>1</sup> would fill this need. In any case, the heart of such a program will be the software engineering course.

The choice of the subject matter of the courses, their durations, and objectives will, naturally, reflect the overall program goals constrained by time, budget, and other resources. The technical courses should provide the background to enable students to understand the nature of a typical company software system development problem. The programming course should imbue the student with an appreciation of the inherent value of modular programming as well as provide him with the tool. The software engineering course must introduce the student to the concept of solving a problem by breaking it up into smaller, simpler problems (i.e., analysis). The student should learn the value of developing a detailed logical design prior to writing any code. S/he should develop an appreciation for the problems of software project management through the study of such concepts as the software system development life cycle, its associated documentation, quality assurance, configuration management, software testing, design reviews, etc. This understanding of the management implications of project development should make the student more amenable to being managed.

Instructors for all courses should be experienced and highly trained (probably masters level) in their respective subjects. Software engineering instruction poses unique problems. There are few experts in the subject and they spend a lot of time disagreeing on many issues. The software engineering instructor should, ideally, be an experienced software project

manager who has become educated in the modern techniques of software development. Since such persons will be hard to come by, an acceptable compromise is a qualified (advanced technical degree) employee with a few years of company software development experience and the motivation to quickly become expert in modern methods of software project development and management.

There are several methods other than the usual instructor lecture which could be effective in a software engineering course. Many of the management topics could be effectively addressed by experienced company personnel acting as guest lecturers. There are excellent video tape courses available on software engineering. There are also several companies offering seminars on various related subjects.<sup>2</sup> While these are expensive, they are also quite informative and, in a well-planned program, could be cost effective. A crucial adjunct to lectures is "hands-on" experience. The students should be required to participate in the analysis, design, documentation, and management of a typical company software development effort. The requirement that such development be a team effort is important. A lack of understanding of the necessity of and problems associated with a team development effort appears to be a major weakness of the typical new college graduate.

The choice of a suitable class project presents some problems. A major lesson to be learned from such a project concerns the amount of paperwork associated with it and an appreciation, from a managerial point of view, of the necessity for this documentation in a large project. The choice of a small project which might be completed in, say, a few months hardly justifies the required amount of documentation. On the other hand, a more typical problem might have to be abbreviated to the point of becoming totally unrealistic and, therefore, of limited benefit. A solution is to choose a problem of moderate size (perhaps a one or two year life cycle) and only concentrate on the conceptual and development phases. After all, it has been pointed out that errors committed in these phases are typically not discovered until after coding has been completed and are the most costly to fix [7].

Student work on the problem should be held as closely as possible to the realities of work on a typical company project. The students should be exposed to the same types of tools, reviews, audits, walkthroughs, etc. that are documented in the standards and practices of the company. In particular, every student should be required to give oral presentations before experienced software engineers and managers who are unafraid to ask probing questions or reveal design weaknesses. Students should also be exposed to management decisions and dilemmas.

#### IMPLEMENTATION

It is clear that the training program implied by the above paragraphs is a lengthy one. It is unrealistic to plan it as a program of part-time study. It is, instead, clearly a full-time program lasting several months. The student would retain company employment and full salary while participating. The company would, if necessary, handle the placement of students upon graduation.

<sup>1</sup> Ada is a registered trademark of the U.S. Department of Defense.

<sup>2</sup> Integrated Computer Systems and Yourdon, Inc., for example.



The facilities required for implementation of the implied program are actually rather modest. Floor and office space will be required to support a group of, perhaps, six full-time instructors, a senior instructor/coordinator, an administrator, and clerical personnel. Thus, a total initial staff of around ten persons is implied. A classroom equipped with chalk boards, screen, overhead projector, a video tape machine and large enough to seat the class comfortably is required. There should be computer facilities available for the students supporting whatever language is being taught. Smaller meeting rooms will be needed, for individual project teams. The classroom may be used for the oral presentations if it is large enough. If not, such a meeting room will be required. An extensive reference library will be desirable. Word processing and/or secretarial support will be necessary to handle the required documentation associated with the class project. Textbooks, course notes, video courses, microprocessors, and other training aids would be provided by the company. Financial support should be available to keep instructors informed of current happenings in academia, government acquisitions, and industry by encouraging attendance and participation in state-of-the-art seminars and short courses.

The potential benefits of such a program are obvious. The availability of a steady supply of software engineers will partially satisfy a company need. In addition, the knowledge that such training exists may well prove an inducement in attracting and retaining new talent. The methods and standards taught in the training program can eventually establish themselves throughout the company. The training program, as it matures, will be in a position to monitor the success or failure of specific techniques. Such research would not only be of tremendous benefit to the parent company but to the software development industry in general.

A clear problem with the proposed program is the fact that it involves a considerable investment and, therefore, firm executive support. Such support may be hard to win without a proven track record to back up claims of potential benefits. Such a complete track record does not yet exist. However, at least one large aerospace company, Lockheed Missiles and Space Company, Inc., Sunnyvale, CA, has instituted such a program. It is now five years old. The remainder of this article will be devoted to a description of this program and lessons learned.

#### THE DSDD PROGRAM

The program, which has been active since late 1978, is known as *Extra Prime Skills—A Data Systems Design and Development (DSDD) Training Program*. It was established to help fill the recognized company need for software engineers and to provide an alternate career path for some of its employees, primarily scientists and engineers. Employees with a technical degree or equivalent background and at least a year of continuous employment with the company are eligible to apply. Applicants are interviewed by DSDD management. Characteristics which the interviewers look for are genuine interest in a career in software development with the company, willingness to accept the implied work load, and the ability to work well in, and contribute to, a group effort. A class of 25 students is chosen from the applicants. Two such classes

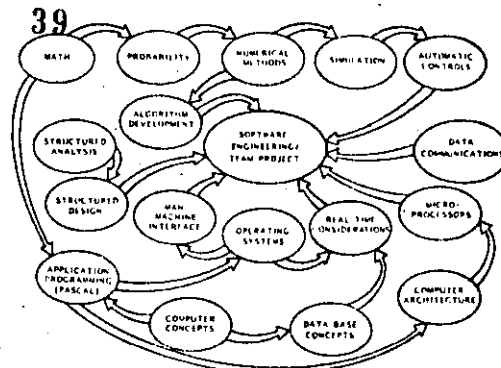


Fig. 1. Support relationships of the various DSDD courses.

are graduated each year. Students are required to attend classes full time, 7:30 until 4:00, Monday through Friday, for the full six months of the program. They have no other company related duties during this time. They continue to receive their normal salaries during the full training period.

A typical student day consists of four to six hours of lecture and two to four hours of study time which may be used for working on homework, computer programs, or the class project. Experience has shown that the forty hours per week on site must typically be augmented by ten to twenty hours, or more, of additional work on the student's own time. Individual classes are taught in blocks ranging from one to six weeks, two hours per day. The exception is the software engineering block which runs continuously throughout the entire six months. Courses consist of lectures by instructors, guest lecturers from within the company, professionals on video tape, and outside consultants who conduct seminars on specific topics. Students are required to complete homework assignments, including computer programs, and take periodic examinations in the various courses. Textbooks and/or lecture notes, at no expense to the student, are provided for each course.

Since most of the software developed within the company is of a highly technical nature, DSDD students receive courses in mathematics and engineering disciplines as well as those in software development. Not surprisingly, the software engineering course is the heart of the program. All other courses support it either directly or indirectly. The relationship of the courses to one another is depicted in Fig. 1.

Fig. 2 illustrates a typical DSDD schedule.

The staff required to support this effort currently consists of five instructors, one senior instructor/coordinator, one administrator, and one secretary. Typical instructor backgrounds include mathematics, computer science, electrical engineering, and other technical fields. A candidate for an instructor position is expected to have a degree (preferably advanced) in a technical discipline. S/he is expected to have experience in the subjects to be taught as well as some general teaching experience. Each instructor is assigned to teach courses amounting to roughly 100 hours of in-class time per six months. During the periods when the instructors are not participating in daily instruction, they are encouraged to at-

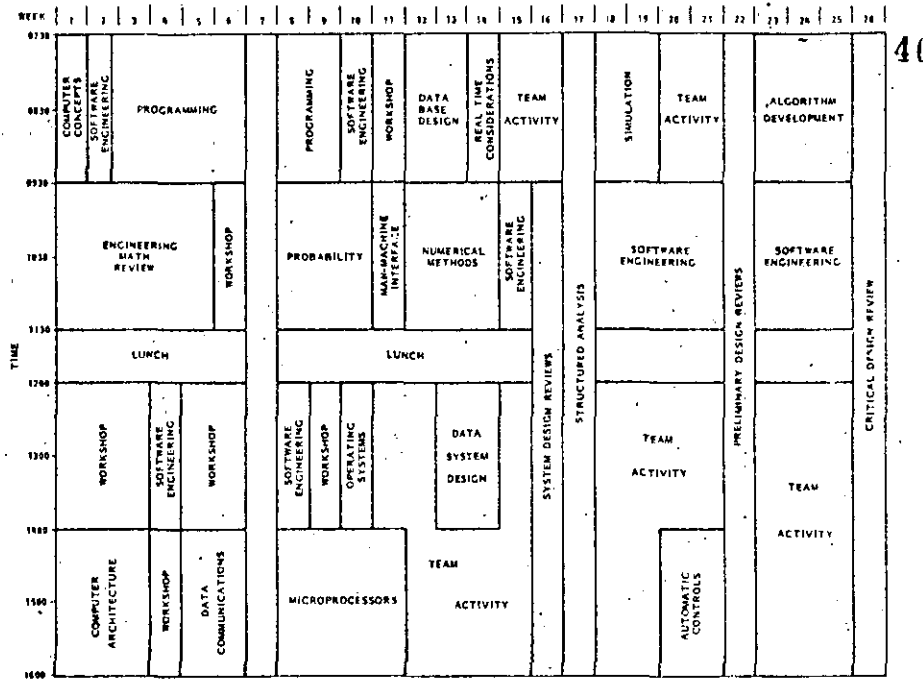


Fig. 2. Typical 26-week DSDD course schedule.

tend other lectures and seminars for the purpose of increasing the breadth and depth of their backgrounds. Sufficient travel money is budgeted to allow each instructor to attend several seminars each year.

Instructors are encouraged, in particular, to keep their respective courses current. This is especially important in a subject such as software engineering which continues to evolve rapidly. The DSDD software engineering course has matured considerably in the last four years. Instruction currently combines lectures, both live and on video tape, special seminars and a lengthy class project.

**THE DSDD SOFTWARE ENGINEERING COURSE**

Topics covered in the software engineering lectures include a brief history of software development and the "software crisis," requirements analysis, software design, software testing, quality assurance, configuration management, the development life cycle, DoD procurement procedures, and the associated documentation.

The class project involves developing the application software to control the operation of a Wind Energy Generation System (WEGS) which is to provide electricity to a number of consumer areas. The students are provided with initial documentation specifying the requirements of the system from which are to be extracted those requirements which might properly be allocated to software. They are also provided with detailed requirements regarding the documents which they will prepare and deliver to the "customer."

The initial work consists of analyzing the system require-

ments and preparing for a formal presentation at which these requirements are reviewed for clarity and completeness and any proposed changes may be presented. This initial presentation is a System Requirements Review (SRR) and is held before a "customer team" made up of experienced company software developers and managers who are selected for this activity. This initial work is done under the supervision of a student committee selected by the staff. The successful completion of the SRR results in the establishment of an initial, agreed-to, or "baselined," set of system requirements from which subsequent work may proceed. Since instruction is heavily slanted toward software development for the DoD, the SRR and all other reviews, as well as all the documentation, are prepared and presented in accordance with the appropriate military standards. Since the SRR is conducted fairly early in the training program (after approximately six weeks) the presentations are typically naïve. Emphasis is placed on presentation style, conduct of the review, and a demonstrated understanding of the nature of the problem. The students are also required to address how they intend to manage the development of the proposed software. Many questions are asked by the customer team for the purpose of pointing out areas where more attention is needed and where expressed ideas are clearly infeasible.

A typical SRR lasts about two hours. The subsequent reviews generally take longer. After SRR, the student committee is dissolved and the entire class is divided up into five 5-person teams. The teams and their respective chairpersons are appointed by the staff. The members of the original

40

student committee are disbursed among the five teams and are not, generally, permitted to act as chairpersons of their respective teams. This gives others a chance to be exposed to the problems of management. The new teams proceed, competitively, to develop a top-level partitioning of the problem and to allocate the baselined requirements appropriately. The results of each team's analysis are presented at a System Design Review (SDR). At the review, each team is required to present its top-level breakdown of the problem into roughly autonomous subproblems, each of which will ultimately evolve into a manageable piece of software called a Computer Program Configuration Item (CPCI). The proposed allocation of requirements to CPCI's is also presented.

In addition, each team is required to develop supporting documentation. This documentation consists of an allocation document detailing the allocation of the software requirements to the proposed CPCI's; a Computer Program Development Plan (CPDP) detailing how development of the software will be managed, and an Interface Management Document (IMD) which serves as a repository for detailed definitions of data and control items crossing interfaces between pieces of software, between software and hardware, and between software and humans. This implies a total of 15 documents from the 5 teams to be reviewed by the staff and customer team prior to the SDR. Since a typical SDR lasts around three hours, three days are generally set aside for the completion of all five of them. After the completion of the last SDR, one team's design and management approach is chosen as the one exhibiting the "least risk" and that approach is then adopted by the entire class for the remainder of the program. The selected team is designated the "integrating contractor" responsible for coordinating the activities of the other four teams. Each team, at SDR, is expected to address potential problems of this upcoming management activity and present its plan for ensuring a smoothly coordinated, post-SDR development effort. The students generally exhibit, during this management portion of the SDR, a greatly improved understanding of, and appreciation for, the problems and benefits of a team effort.

After SDR, the class has approximately 5 weeks to prepare for their Preliminary Design Review (PDR). The PDR generally consists of presentations by each team on its assigned CPCI. By PDR, each CPCI will have been further broken down into functional components with previously existing and newly derived requirements allocated appropriately. The integrating contractor generally addresses matters of management, perceived areas of risk, requested baseline changes, etc. The PDR generally requires one to two days to complete. The teams are required to produce development specifications in accordance with the appropriate military standard. The integrating contractor is responsible for updating the CPDP and the IMD. After the PDR is completed, one CPCI is selected and the class works on converting the functional analysis into a physical design. The functions of the CPCI are gathered into functionally cohesive Computer Program Components (CPC). The results of this effort are the subject of the final review.

The last month is spent preparing for the final review. This is the Critical Design Review (CDR). It typically requires one day and, therefore, is often held the day before graduation.

By this time, the teams have developed the CPC's of the selected CPCI to a codable level. The required documentation is a Product Specification, which describes the physical implementation of the preceding analysis, an updated CPDP and an updated IMD. By CDR, the students have generally become quite comfortable in their role as contractor and typically conduct a very professional review.

#### THE CUSTOMER TEAM

41

The role of the customer team in the reviews is crucial. They rely on their experience and knowledge of the class project to help them judge the quality of the reviews. Severe and unremitting criticism has produced an early defeatist attitude which has seriously degraded the value of subsequent instruction. On the other hand, conducting reviews before a realistic customer is probably the most valuable experience of the entire program. The customer team, therefore, "plays its role" by asking penetrating questions, requesting action items, and openly commenting about design or management features which they find troublesome. The proper mix has not been easy to find and requires iteration. But it is too important to ignore. The customer team has generally been experienced in all phases of software development, including software management, and most have attended several actual reviews with typical company customers and, in some instances, have actual customer experience. At least one team member has repeated this experience with every class to date. This has had the benefit of providing continuity in customer attitudes and role-playing. It has also been found beneficial to have former students on the customer teams. The comments of graduates are typically of special interest to the students.

#### AN EVALUATION

The program, as described in the above paragraphs, clearly represents a major commitment on the part of the company. The annual budget for DSDD is approaching 1.5 million dollars. This is, of course, overhead money and, therefore, clearly implies that DSDD enjoys the support of the highest levels of management. The continued existence and evolution of DSDD reflects the conviction that the work being done is both important and successful.

The ability of any such program to evolve is, of course, one of its most important characteristics. DSDD courses are constantly being modified as instructors find better ways to teach specific areas. In addition, surveys are sent out periodically to former students and their managers. These surveys provide some feedback on student performance and the usefulness of the current courses offered. They have revealed areas of strength and weakness. Fig. 3 and Table I summarize a statistical breakdown of recent survey responses.

Of primary concern is how immediately useful a typical student is after leaving the program to join an existing development project. The data collected so far are too scant and immature to support any firm conclusions. However, at least one manager is on record as having observed that his DSDD graduates seem to have roughly a two year head start over new college graduates with technical degrees. In addition, they are very stable employees. In an environment of high mobility and turnover [9], less than 5 percent of all DSDD graduates have

- JOB CATEGORY**
- A. DATA SYSTEMS CONCEPT DESIGNER
  - B. SOFTWARE DESIGN SPECIALIST
  - C. COMPUTATIONAL HARDWARE
  - D. SOFTWARE DEVELOPMENT SPECIALIST
  - E. DATA SYSTEM INTEGRATION & TEST
  - F. DATA SYSTEM OPERATION
  - G. CONFIGURATION CONTROL & DOCUMENTATION SPECIALIST
  - H. OTHER
  - I. NONE OF THE ABOVE

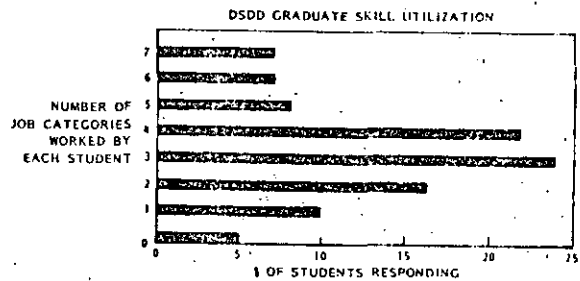
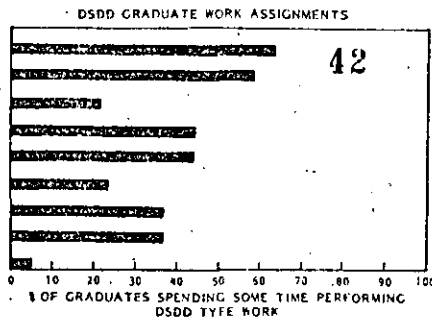


Fig. 3. Summary of survey data on DSDD graduate utilization by job category.

TABLE I  
SUMMARY OF SURVEY RESPONSES

Subject	Utilization			
	Never	Seldom	On Occasion	Often
Calculus	362	272	132	42
Probability	49	36	10	6
Control Theory	55	29	14	6
Data Communications	24	21	31	23
FORTRAN Programming	43	24	21	11
PASCAL Programming	66	35	13	6
Numerical Analysis	84	21	14	0
Data Base Concepts	17	21	34	24
Simulation	48	30	16	7
Algorithm Development	24	35	21	10
Technical Writing	10	3	36	57
Software Engineering	12	6	28	50
Structured Analysis	8	18	37	37
Structured Design/Programming	17	18	31	34
Real Time Consideration	20	31	27	22
Microprocessors	29	22	25	24
Operating Systems	20	27	37	17
Decision Analysis	26	24	35	15
Display Sys/Man-Mach. Interface	24	13	32	30

Did you enjoy DSDD? Yes 965 No 51 Would you do it again?  
 Yes 902 No 102

If you are doing programming, has the DSDD training an aid in producing programs. Yes 851 No 172

What programming language is the basis of the work you are doing and did DSDD aid you in that work. (78 responses)

ATLAS 62, PASCAL 91, FORTRAN 162, Assembly 142, Basic 102, Machine 31, Cobol 124, Soviet 11, C++ 11, PL/M 11, JCL 11.

Has M/M training an aid? Yes 412 No 162

Were you able to exactly integrate into the organization to which you were assigned? Yes 812 No 192 (If no, please comment)

left the company. This is even more significant in light of the fact that there is no postgraduate company service requirement.

The DSDD graduates serve the important function of carrying their knowledge to their respective projects. This

has two benefits. First, it immediately provides dissemination of new and useful information to co-workers who did not have the opportunity to attend classes. Second, the knowledge and documentation that is making its way into the field may slowly establish consistent standards and methods of software development throughout the company. The students, furthermore, are not just "software people." The technical courses enable them to converse with professional scientists and engineers in technical terms, thereby reducing the hazards of misunderstanding which generally plague human communication. In short, DSDD has evolved into a strong, successful program.

More important than obvious successes, however, are the areas of possible weakness which are found. Identification and elimination of these will guarantee continued improvement in the program. Some of these areas will now be addressed.

The DSDD mathematics courses have included reviews, lasting three to six weeks, of the calculus, statistics and probability, and numerical methods. These courses are viewed as essential to the objectives of the program. However, because of differences in the backgrounds of the students, the level of enthusiasm for these courses varies widely. Part of the solution to this problem lies in the careful selection of in-class examples and homework problems which clearly illustrate the relevance of these courses.

The student project is designed with two specific goals in mind. First, it is complex enough to require a team effort and, therefore, a realistic management effort. Thus, it gives students exposure to the methods and problems associated with software development and software project management. Second,

it involves a problem requiring some technical sophistication on the part of the students and, hence, allows application of some of the skills developed in other courses. But the class project suffers from the obvious constraints of time and resources. It was pointed out earlier that the project should be of sufficient complexity to justify the required documentation. The project satisfies this requirement. As a result, only part of its total life cycle is ever addressed. The students only take the project as far as CDR which precedes actual coding. Thus, the students develop detailed modules which will never be coded, test plans which will never be implemented, etc. In addition, certain time-consuming aspects of the project have been short circuited by allowing the students to make some simplifying assumptions. While these unrealistic elements do not negate the tremendous worth of the project, they have the potential to lessen its impact. This problem is diminishing with each class, however, as minor adjustments are continually made to infuse the project with more realism.

Former students complain of a resistance to change on the part of their co-workers and managers. Management feedback tends to substantiate this. Many existing managers are only too aware of the software crisis and the difficulty in managing a software development project. They are understandably wary of new techniques in a field where cost and budget overruns are common, especially in the absence of detailed data showing predictable increases in productivity linked to these methods. Such data will be forthcoming eventually. Until that time, it must be recognized that managerial pragmatism is proper and will be overcome only slowly as the techniques prove themselves and evolve to fit the manager's respective contexts. As the hard evidence accumulates, however, it may be assumed that the drive toward universal acceptance will greatly accelerate. The students are, therefore, urged to adapt and use the ideas of requirements analysis, modularity, clear and disciplined documentation, etc. in their own work. In this way, the required track record will be slowly established. Continuing efforts by the DoD to effect better system design approaches will also, no doubt, prod managers to be open to new techniques.

The company assumes the responsibility of placing new DSDD graduates. DSDD policy in this area is set by a guiding board of directors. Every attempt is made to match graduates with open slots in such a way that student desires and company requirements are met. Obviously, there is never a set of 25 perfect matches. Postgraduate surveys have contained complaints in this regard. Efforts to improve the placement process are continuing. The current placement effort begins with the initial candidate screening interview. The prospective student's responses and résumé are used to gauge his/her special areas of interest. The candidate is also informed that the initial postgraduate placement may not be precisely what was desired. Monitoring and counseling of students and monitoring of company needs continues throughout the six months of classes. This effort has greatly reduced the number and nature of the complaints.

The inherent subjectivity of a topic like software engineering can lead to frustration on the part of the students and the instructor. The customer team expresses certain opinions during the reviews, guest lecturers express different views,

and the instructor may end up presenting a third. While all the expressed viewpoints have several common denominators, the students often become concerned over the inability of anyone to point out one specific "right answer" to questions of management, testing, quality assurance, etc. This frustration is mirrored in the instructor who perceives the problem but is unable to fully alleviate it. It is necessary for the instructor to constantly remind the students that there are no "right" answers, merely "less risky" ones as indicated by evidence compiled from previous projects. The purpose of the course is to present methods, not solutions.

## 43

### CONCLUSIONS

The objective of DSDD is to alleviate the software engineering shortage at LMSC by providing experienced engineers with the opportunity to redirect their careers toward software. The feasibility of such a program has now been amply demonstrated. The program is recommended to other companies currently experiencing a shortage of experienced software engineers.

### DIRECTIONS OF FUTURE GROWTH

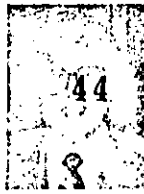
One thing seems clear. The need for DSDD and the necessary support are likely to continue into the foreseeable future. This, of course, implies further growth and evolution. Areas of expansion which are already being actively pursued include the development of abbreviated courses to be offered to a spectrum of employees ranging from managers to new hires. The subject matter will be similar to that of the six month course but greatly compressed. Managers are being offered overview courses in the concepts of requirements analysis, logical design, structured coding, testing, documentation, etc. The purpose of such courses is to acquaint managers with the tools and methodology in which DSDD students are trained and suggest proper ways to use these students after graduation. New hires will be offered orientation training before they arrive at their designated organizations. This training will consist of lectures and short exercises in software engineering subjects, programming, and technical disciplines. Other directions of growth include the following.

- Establishment of improved lines of communication to and from active projects for the purpose of assessing student performance and gathering data on the impact of the new techniques.
- Improvement of existing teaching methods through incorporation of new equipment such as personal computers and simulation, hardware/software.
- Establishment of lines of communication and cooperation with local universities for the purpose of exchanging ideas, data, students, and instructors.
- Replacement of the current Pascal programming instruction with a course in Ada.

### REFERENCES

- [1] H. D. Mills, "Software development," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 265-273, Dec. 1976.
- [2] J. Fagenbaum, "A new breed: The software engineer," *IEEE Spectrum*, pp. 62-66, Sept. 1981.
- [3] J. W. Plummer, "Extra prime skills," *Military Electron./Counter-*

- measures, p. 29, Dec. 1978.
- [4] R. W. Jensen and C. C. Tonies, *Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
- [5] R. W. Jensen, C. C. Tonies, and W. I. Fletcher, "A proposed 4-year software engineering curriculum," *SIGCSE Bull.*, vol. 10, pp. 84-92, Aug. 1978.
- [6] P. Fireman, A. I. Wasserman, and R. Fairley, "Essential elements of software engineering education," in *Proc. Int. Conf. Software Eng.*, 1976, pp. 116-122.
- [7] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226-1241, Dec. 1976.
- [8] C. D. Labelle, K. Shaw, and J. J. Hellenack, "Solving the turn-over problem," *Datamation*, pp. 144-152, Apr. 1980.



James P. McGill received the B.S. degree in mathematics from the University of Maryland, College Park, in 1970 and the M.S. degree in applied mathematics from the University of Nevada, Reno, in 1977.

He undertook additional graduate study in numerical methods from the University of California, Davis, from 1977 to 1979. He joined Lockheed in 1979 after finishing his graduate study and a tour of duty as a Naval Intelligence Officer. After two years of developing simulation software in support of ballistic missile defense research, he joined the DSDD program and became the software engineering instructor.

## Managing Software Engineering Projects: A Social Analysis

WALT SCACCHI, MEMBER, IEEE

**Abstract**—Managing software engineering projects requires an ability to comprehend and balance the technological, economic, and social bases through which large software systems are developed. It requires people who can formulate strategies for developing systems in the presence of ill-defined requirements, new computing technologies, and recurring dilemmas with existing computing arrangements. This necessarily assumes skill in acquiring adequate computing resources, controlling projects, coordinating development schedules, and employing and directing competent staff. It also requires people who can organize the process for developing and evolving software products with locally available resources. Managing software engineering projects is as much a job of social interaction as it is one of technical direction. This paper examines the social arrangements that a software manager must deal with in developing and using new computing systems, evaluating the appropriateness of software engineering tools or techniques, directing the evolution of a system through its life cycle, organizing and staffing software engineering projects, and assessing the distributed costs and benefits of local software engineering practices. This purpose is to underscore the role of social analysis of software engineering practices as a cornerstone in understanding what it takes to productively manage software projects.

**Index Terms**—History of software engineering, organizational impact, social analysis, software engineering project management, software life cycle.

### I. INTRODUCTION

MANAGING software engineering projects requires an ability to comprehend and balance the technological, economic, and social bases through which large software systems are developed. This is necessary to produce usable systems. This paper examines how people organize and perform the

engineering of software systems. The focus is on assessing the current understanding of the social aspects of software project management. Therefore, this material will directly relate to, if not overlap, technological and economic aspects of software engineering project management presented elsewhere.

Software systems are developed, used, and evolved by people in a variety of organizational settings. These people are identified by their skills, tasks, and profession as well as their conception of what work needs to be done. Software engineers, programmers, systems analysts, software project managers, user-specialist liaisons, instrumental and clerical users, user department managers, vendor representatives, contract monitors, and others all can become involved in moving a system through its life cycle. The settings where their work takes place also varies as will the centrality of local software engineering efforts. These often vary according to the kinds of 1) products produced, 2) computing technology in use, 3) production problems solved (or created) with software applications, 4) supporting vendors, and 5) public accountability [23], [37]. The variations of these factors characterize the increasing complexity of organizational arrangements where software systems are managed.

One basic question of interest is to what extent do these organizational arrangements impinge on local software engineering and project management (SEPM) practices? Certainly, if these arrangements have little or not effect on local SEPM, then we can safely ignore them. However, if they have some effect, how significant will they be? How will they constrain the ease with which reliable, useful, and maintainable software systems can be produced? This paper will show that a variety of social arrangements intimately shape both the course and the outcome of local software engineering activities as well as how they are managed. Thus, the purpose is to underscore the

Manuscript received May 4, 1982; revised November 16, 1983.

The author is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

role of social analyses of software engineering practices as a cornerstone in understanding what it takes to manage productive software projects.

45

The remaining sections of the paper review the current understanding of the social aspects of software engineering and project management. The next section outlines assumptions and define concepts that form the basis of contemporary social analyses of computing [22]; Section III examines the history of social arrangements in software engineering projects. From this, a set of recurring social situations are identified in Section IV that give rise to major problems in engineering software throughout its life cycle. Next, a set of underlying relationships are described in Section V which provide an account as to why these problems occur. Section VI outlines a set of strategies for managing the social arrangements that affect local SEPM practices in line with the emerging understanding of the future directions of software engineering. Finally, conclusions are drawn that substantiate the need for more social analysis of software engineering and project management.

## II. ASSUMPTIONS AND DEFINITIONS

The starting point for any activity requiring formal management is identifying the basic tasks to be performed. These tasks are planning, organizing, staffing, controlling, directing, coordinating and scheduling. Descriptions of what these tasks entail appear in introductory management texts and more recently in software management tutorials [36]. This set of tasks implies that the work of engineering and managing software projects is to be done in careful *order*. In particular, many analysts use some model of the system life cycle to denote the process producing that order. "Waterfall charts" and evolutionary "S-curves" are common representations of these software life cycle models. Accordingly, a software system is specified before designed, implemented after designed, tested, and documented, put into operation, and then maintained until either converted, replaced, or unused. Managing the orderly production of a software system throughout its life cycle becomes the process that must be planned, organized, staffed, and so forth. But how is that order achieved and how stable is it? The concern here is with the recurring situations that disrupt and put that order into flux. When these *problematic situations* arise, the order must somehow be realigned and re-established. As this paper shows, problematic situations are common in SEPM.

If there is a single thread that links many social analyses of computing, it is a focus on examining apparent patterns of problematic situations attendant to computing in various settings [10], [17], [18], [20]-[27], [31], [37], [38]. For SEPM, this means identifying those situations that disrupt the expected orderly engineering and management of various software life cycle activities.

It is often straightforward to identify conditions that give rise to problematic situations in complex settings. First is the complexity of the setting itself as outlined in the introduction. People work with computing systems in different kinds of settings. Next, each participant has his/her own agenda of what work needs to be done, how to divide the work, what tools to use, and who to turn to when problems arise. Subsequently, part of a software project manager's job is to coordinate the

agendas of participants who become involved with system development. Unfortunately, conflicts can arise between participants with different agendas that are not always easy to resolve.

Conflicts may occur when different participants, say software designers and instrumental users, have an interest in getting things done their own way. For example, who determines how systems should be designed? The software designers, the users, or both are all appropriate answers at different points during system development. But where or how do the boundaries get drawn? Such a dilemma might be readily resolved in most instances, but what happens when participants are uncertain about the consequences of their actions? As Professor H. A. Simon observed long ago, such decision making often occurs without complete information: information is a scarce and costly resource [40]. Thus, participants become justifiably concerned over *uncertain conditions* that can jeopardize the performance of their work: 1) mistakes they might make and be held responsible for; 2) unexpected delays that hinder the timely completion of tasks; 3) discretionary authority others can use to constrain behavior or define work content; 4) use of ambiguous criteria to evaluate the quality of an individual's or group work; and 5) ill-defined procedures to follow when developing a system with vague or shifting requirements. Conditions such as these give participants reason to follow a course of action more closely aligned to their *interests at stake*, available *opportunities*, and *constraints* they perceive when unresolved conflicts persist. But at the same time, as participants act in an uncoordinated way to mitigate these conditions, their actions can redistribute the brunt of uncertainty onto others who in turn may act in similar ways. Thus, as participants encounter and act toward uncertain conditions arising during system development, they may unintentionally act to make their situation more problematic.

In settings with a large number of participants whose work flows are interdependent and highly specialized, the regular performance of software engineering work gives rise to problematic situations. Accordingly, the organizational climate and the productive flow of work will depend upon how these situations are handled. Thus, it is not surprising to see that project managers spend most of their time negotiating in face-to-face meetings with various participants making sure work is progressing, in "putting out fires," and in acquiring and distributing access to available supplies of engineering resources [9], [19], [25], [32], [37]. This *negotiation work* is a key social component in managing the course of a software engineering project. Unfortunately, it is not well understood or appreciated except by those who have managed one or more large-scale system development projects.

Software engineering is concerned with the development, use, and evolution of software artifacts. These artifacts are inherently social objects: people find meaning and value the creation, use, and evolution of software modules, system designs, user manuals, etc. The life cycle of a software system must be understood not only as a mathematical or technological endeavor, but also as the outcome of a complicated social process. Software engineering projects take place in complex organization settings. The factors that characterize the process of system development within a specific organization, the

intended order to engineering software products, the various agendas that participants try to enact, and other uncertain conditions that become problematic shape the social context for local SEPM. Thus, software project managers must negotiate project planning, organizing, staffing, and so forth through a web of *circumstantial* social arrangements.

The next section examines software engineering practices that in some way deal with circumstantial arrangements in organizational settings. One important distinction made in reviewing these studies is the extent an analyst views the web of social arrangements where software engineering work takes place as being *separable* from the tools and techniques of software engineering. In short, what features of the work setting does the analyst bring into account and what is ignored? As the next section shows, the assumed coupling of the general practice of software engineering with the ongoing computing work occurring in a setting determines the quality of the social analysis performed as well as the outcomes or insights possible.

### III. A HISTORY OF SEPM PRACTICES

The ability to comprehend current SEPM practices is determined in part by what we know about previous experiences. However, there are a few studies of early and recent software engineering projects that specifically address the social dynamics of those endeavors. Thus, part of the challenge is to sift through published studies to find those which provide observations substantial enough to build an historical understanding. In particular, the focus is on studies that point to "what to manage" during a software engineering project. The starting point is the SAGE project, the first large-scale software project begun during the mid-1950's.

Early experiences with large-scale software development appeared during the construction of the SAGE air defense system [13]. SAGE was also the first of many large-scale, complex ground-based command and control systems. The concerns for SEPM that emerged during this project included: 1) control—make sure that programming occurs after system requirements are established and no sooner; 2) flexibility—recognize that system requirements change and that systems need to be portable to different settings; 3) people—differ in capability, and 4) management—be able to direct many (>100) people organized in complex team structure, keep them motivated, etc. [16]. Thus, the established focus on managing software projects was on the difficulties of managing software *development*, not its use or evolution. Hosier's study [16] of the pitfalls and engineering safeguards practiced during the SAGE project made clear that managing software development was a big problem.

During the 1960's, a number of software projects comparable in size to SAGE took place. Large-scale systems such as the SABRE airline reservation system, the IBM OS 360 operating system, the CTSS and MULTICS operating systems, and the Manned Space Flight systems represented major software projects of this period. Although people working on these projects were able to learn from the SAGE project experiences, project managers reported a growing set of dilemmas in SEPM. These included 1) system requirement specifications were still vague although more formal and voluminous, 2) or-

46

ganizations cannot easily substitute staff for time, 3) productivity measures such as "man-month" are inappropriate and misleading, 4) systems undergo substantial rewrite between releases—systems must be built to evolve, 5) projects pass through generations of equipment, staff, vendors, and sponsors, 6) commitment of staff to a project's success is necessary to achieve such an outcome, 7) choice of programming languages affects project productivity, 8) system designs generally reflect communication structures within the organization, 9) variations in the skill of programmers is substantial, and 10) operation differences exist between organizational, managerial, and technical criteria for project success [7]-[9]. This set of observations viewed in light of the apparent technical difficulties in developing reliable, easy-to-maintain systems was what some people then called "the software crisis."

One concern repeatedly expressed during the first two international workshops on software engineering in the later 1960's was how to improve software management [8]. Most suggestions that appeared, mainly represented a commitment to investigate new software technologies: time-sharing operating systems, high-level programming languages, mass-produced software components, structured programming techniques, and software cost/price mechanisms. That is, the way to improve software management is through new software technology. But how these technologies help software managers resolve the SEPM dilemmas they faced was unclear or missing. Thus, strategies for managing the range of dilemmas representing the software crisis were somehow displaced unless addressable through new technologies.

Additionally, studies of software engineering practices in smaller, more medium-size projects were noticeably absent during this time as were studies describing the ongoing use and maintenance of the large-scale systems mentioned above.

The early 1970's were marked by the publication of results of the CCIP-85 study [3], [28], [45]. This now famous study of the U.S. Air Force's information processing requirements in the 1980's revealed that software costs as the percentage of total system costs were substantial and would soon dominate. Rising software costs were the problem, and strategies for improving software productivity became the focus. Accordingly, improving project organization and management was among the prescriptions offered to reduce software costs and increase productivity [3].

Improving project organization and management became synonymous with developing a project organization structure that would provide increased managerial control and coordination. Two popular alternative forms emerged; "egoless programming" described by Weinberg [44] and "chief programmer teams" due to Baker [1]. Weinberg's approach sought to open-up interaction and diminish conflicts over control, coordination, and expertise for people in a project team. Software people satisfied with working conditions and other team members would be easier to manage and would develop more reliable systems. However, Pettigrew [34] and Danziger [10] found that similar conflicts also occur between people in project teams and user departments. But proposals for egoless computing or egoless work organization never appeared. Baker's team structure, on the other hand, was founded on a disciplined management approach where conflicts or un-



certainties over what software work was to be done are mitigated through formal lines of authority, control, and expertise established with the team. Kraft's [29] elaborate response to Baker's proposal contends that Baker's proposal represents a troublesome reapplication of the principles of "scientific management" intended to deskill and downgrade the work of software professionals as well as encourage status-based segregation of project participants. Clearly, some organizational structuring helps diminish certain conflicts and improve project management. But, what was needed was a rethinking of the organization of software project members in terms of the work they do, what stake they have in it, with whom they interact within the local setting, and what resource constraints they encounter in their work rather than simply proposing incremental variations on the team structures put forth by Weinberg and Baker.

By the mid-1970's, Boehm's [4] seminal paper made clear that interest in understanding software engineering was bound to the activities occurring during the life cycle of a software system. Although concern for the life cycle of complex systems existed prior to this, the software life cycle provided a unified view of the range of activities organizations had to engage and manage to produce large software systems. This breakdown also made clear that software development costs were not uniformly distributed throughout the system life cycle. In particular, testing and maintenance (i.e., keeping a system usable) dominated all other development activities [4]. Subsequently, SEPM research activities again focused on 1) how to improve the rate of software production during each life cycle activity through new software technologies as well as 2) how to reduce system life cycle costs through structured project management [35]. But what activities or situations drive software life cycle costs? How should software development be managed to minimize life cycle costs and boost productivity? These became two guiding questions which most present-day software engineering activities now address. To no surprise, these are related concerns.

The current understanding of what drives software life cycle costs comes from two lines of research. The first focuses on software problems found either in the delivered software product, or in the software production process. The second focuses on the conditions which give rise to these problems in the *consumptive* or *productive unit*: the organizational work setting.

In the first, the quality of software is limited due to a number of problem areas. These problem areas are software cost drivers. The most frequently cited cost drivers include: 1) errors in software specification, design or implementation, 2) system requirements that change during development, 3) inconsistent and incomplete specifications, 4) a general lack of adequate software development tools and production methodologies, 5) inadequate system verification techniques, 6) communication problems among users and system designers, and 7) recurring neglect for software maintenance. Subsequently, new software specification languages, analyzers, methodologies, and related technologies were developed targeted to the first five problem areas. However, few convincing studies of the efficacy of the new technologies in mitigating software production costs are yet to appear. Communication

and maintenance dilemmas, on the other hand, were yet to receive as much attention as "improved technology" solutions.

In contrast, the second overlapping line of research emerged focusing upon dependencies between how software products were produced and consumed in different organizational settings. In software production organizations, major problems centered around 1) division of labor,<sup>1</sup> 2) continuous modification of systems, 3) growth and servicing multiple system versions, 4) increasing formalization of engineering procedures and plans, and 5) lack of ability to externalize product knowledge [2].<sup>2</sup> Similarly, there was often internal organization pressure to 6) coordinate product development and marketing activities, 7) respond to changes in the organization's environment, 8) maintain commitment to tight production schedules, and 9) perform within local budgetary or economic constraints [2], [5]. Further, the production of requirements analysis, software specifications, design, testing, integration, and maintenance were typically performed by average (not expert) programmers and managed by other technical specialists who had ascended into management positions [5], [24]. Although the interaction between these problems was unclear, it seems likely that this set of conditions could easily give rise to the problematic situations that plague software production [42]. It also appears that similar conditions affect hardware and VLSI system development [19], [38], [39].

In software consumption (user) organizations, many related concerns surfaced. Software systems were found to be more difficult to use when users did not participate in system specification or design [31], and when systems were more machine-oriented rather than user-oriented [21]. System designs were seen to reflect not merely the communication patterns within an organization, but also its political order [10], [18]. The cooperation of systems developed by different organizations could be difficult to comprehend for users in spite of each system being acceptably well documented [33]. Software maintenance activities were found to be primarily driven by user requests for additional system capabilities (e.g., produce new reports) and to a lesser extent by performance improvements, bug repair or other specialist enhancements [30], [41]. New software tools were observed to have potentially significant hidden costs in how they were applied that could exacerbate the maintenance of embedded applications systems [24], [37]. Further, accounting for the costs and benefits of software system use was shown to be difficult, poorly understood, and often biased according to participants' self-interests [20], [25], [26], [37]. The life cycles of local systems were found to be layered and intersecting one another [26]. Finally, the set of activities occurring during a system's life cycle were shown to be embedded in and shaped by the prevailing social order within its organizational setting [26, 37]. In sum, these studies began to outline

<sup>1</sup> Division of labor refers to the problem of how to divide system development work between systems, staff, and one another given the number and skill of staff members working with locally available computing resources.

<sup>2</sup> "Product knowledge" is the understanding of how programs work individually and in cooperation with each other. This exclusive knowledge is generated during software development. "Process knowledge," on the other hand, is about designing, coding, testing, and interpreting programs, as well as managing these activities [2].

the range and distribution of social arrangements that drive the costs of software consumption.

Most present day strategies for managing large software projects focus on applying large tools or techniques to further structure, rationalize, and automate software production [11], [12], [35]. As such, these recommendations need to be put into practice according to policies, procedures, standards, review boards, change control committees and other administrative mechanisms that constrain project budgets, plans, schedules, benchmarks, and reporting activities [11]. In short, these recommendations imply that more organizational bureaucracy is needed to ensure the routine (i.e., predictable) life cycling of local software systems. However, as the routine becomes more apparent, it may be automated to reduce costs, boost production, and further embed bureaucratic mechanisms. But, what is missing are recommendations for how to manage the plethora of problematic situations in life cycling software in ways that are not amenable to relief through technological advance and subsequent bureaucratic proliferation. Whether the costs of cycling large software systems or the ease of managing software engineering projects can be achieved other than through increasingly automated, bureaucratic means remains an open question.

The next section examines the social and organizational arrangements identified so far as they may appear during a software system's life cycle in order to put forth a set of relationships that characterize the social aspects of software management.

#### IV. SOFTWARE LIFE CYCLE

The starting point for identifying major issues in the social aspects of software engineering and project management is the software life cycle. Although our discussion of the software life cycle is brief, it incorporates the activities of both software production and consumption [26], [37]. This choice allows us to examine the passage of software through 1) software life cycle engineering activities 2) work routines of local participants, and 3) participants moving through local settings [37]. This starting point also provides a continuity with contemporary concerns for what affects life cycle costs and how to manage them. Further, it serves as a point of departure into a summary of the underlying relationships that link local SEPM practice to the patterns of social action around software systems we observe. Each stage of the software life cycle is examined in turn.

*Initiation and Adoption:* A software system is initiated when participants propose and make the decision to adopt it. The decision to adopt is a *decision to innovate* local computing arrangements. The proposed system somehow meets a "need" that existing computing arrangements do not. The need substantiates the decision. However, on closer examination, the possible range of conflicts within commonly identified needs is large. For example, the need for an industrial products manufacturer to acquire a new computer-aided manufacturing system may depend upon 1) overcoming organizational contingencies such as frequent failures or delays with the existing manufacturing facilities, 2) the perceived ease with which manufacturing activities (or workers) can be better controlled, 3) the apparent accounting benefits arising

from standardized inventory and production control reports generated by the new system, 4) whether manufacturing users are convinced that the new system will make their work more satisfying or entertaining, or 5) possessing a "state of the art" manufacturing system that will help attract or retain talented engineering staff. The point here is not whether all of these needs can be met, but instead, whose agenda are they on, how are they prioritized, who determines the priorities, and whose interests are served when some need is fulfilled.

*Requirements Analysis:* Participants are concerned with two kinds of system requirements, nonoperational and operational. Nonoperational requirements indicate the *package*<sup>3</sup> of resources that the new system assumes must be in place, or are readily available, to ensure its proper operation. Similarly, such requirements may indicate that certain tasks within the local products process be structured to be compatible (i.e., made efficient) with the new system. On the other hand, there are operational requirements for developing a system in terms of its performance characteristics (e.g., response time), standard interfaces, assuring engineering quality practices, portability, user-orientation, and so forth. However, requirements for the system to be cost effective, produced within resource constraints, and be easy to use and manage have both operational and nonoperational implications. But none of these requirements specify what the system's operations are. Instead, they outline the preferences of participants to achieve a certain kind of order through the life cycle of the focal system. These requirements form the criteria for evaluating both the direction and the success of the system development effort. As we saw earlier, evaluation criteria (and thus requirements) are subject to differing interpretations and debate among participants.

*Selection:* Once a binding decision is made to adopt a new system, which system will do the job? Should the system be developed with in-house staff or should it be purchased from an outside vendor? Going with in-house staff facilitates the development of local product and production knowledge useful in system maintenance. But if the system represents an unfamiliar or unproven technology, uncertainty over completing the project within resource constraints may point to a lack of incentives for an in-house effort. On the other hand, going with an outside vendor means trying to figure out the strength of the system's signal from the promotional noise. What criteria can be used to filter promotional information on the new system: 1) vendor reputation, 2) prior experience of similar users, 3) performance characteristics, 4) quality of available documentation, or 5) ease of fit into local computing arrangements? In any case, uncertainty over what to consider in selecting from whom to get the system is present. Thus, it is very likely that system selection will be influenced either by the mobilization of participants favoring

<sup>3</sup> A computing package consists of not only hardware and software components, but also organizational facilities to operate and maintain these components, organizational units to prepare data and analysis, skilled staff, money, time, management attention, application-specific know-how, staff commitment to modern engineering practices, and policies and procedures for ensuring the orderly production of additional applications [25], [37]. Without some such ensemble of these resources a given system is not usable for very long.

one product or by the participants whose input is trusted most by those making the selection decision.

**System Specification:** What is the system to do? What are the objects of computations and what operations are applied to them? How can these specifications be represented so that either their internal or external consistency, completeness, and correctness can be checked? Clearly, use of software specification languages helps. But who participates in specifying the system? Problems found in specifications may be due to oversights in their preparation or conflicts between participants over how they believe the system should function. Although a software specification language or methodology may serve as a medium of communication among participants, the language does not resolve conflicts that might exist between participants over what the system is to do. But the medium may make the conflicts more apparent. Then, who decides how to resolve a specification conflict, who has a visible stake in achieving a particular outcome, and how will specification responsibilities be divided among participants? Each of these questions point to tacit or explicit negotiations between project participants that must occur in the course of getting system specifications developed. Subsequently, the outcome of these negotiations will shape how stable the specifications will be.

**Design:** Designing a system entails deriving its configuration and detailing the computational procedures and objects from the available specifications. Developing a system's architectural design means articulating an arrangement of system modules that progressively transform the objects of computation into work products based on local computing resources. This articulation includes 1) choosing a system design technique, 2) developing and rationalizing alternative configurations, 3) employing a standardized notation for describing system architecture and interfaces, 4) determining the order of module development (i.e., top-down, bottom-up, hardest first, easiest to test, user interfaces first, etc.), 5) mapping system configuration onto staff to divide the labor, and 6) renegotiating any of these if local circumstances do them in. On the other hand, developing a system's detailed design means articulating the computational procedures organized in the architectural design. This stage of design requires interactive access to user knowledge of work procedures being codified into the system. This knowledge is usually dispersed across many participants with varying degrees of familiarity and commitment to the precision of articulation required for computational codification. Since this knowledge is difficult to access, gather, evaluate, codify, and stabilize, various system designs will be plagued with errors of omission or misarticulation. As these problems thus emerge, system design and possibly the software development artifacts preceding it will be redefined.

**Implementation:** System implementation involves coding the design into a computer-based executable form. Choice of programming language comes into play here, as do techniques for assuring the executable program systematically realizes the system's design, specifications and requirements. However, implementation also includes introducing early versions of the system into the work routines of its users. Much handholding between system specialists and users can take place to smooth the introduction. However, if users hold that the system is

being imposed on them without their earlier participation, then a variety of counterimplementation actions may appear marking their resistance to the system's introduction [17]. Therefore, to ensure the system's integration into the local work settings, participants must engage in a series of negotiations to 1) establish sustained service for the new system, 2) get enhancements to the delivered system to improve its fit, 3) eliminate major system bugs, 4) train new users, and 5) acquire top management support to buffer or delay ongoing production schedules during the transition. But participation of users with the system's developers earlier in the life cycle may obviate the need for these negotiations.

**Testing:** Most system testing is heuristic and generally performed through operation. Formal testing is too time-consuming and not yet widely understood. As a result, the division of labor in testing a system usually leads to software developers performing isolated tests on system components and users discovering problems as the delivered system supports more routine usage. When difficulties ("bugs") appear, a collective effort begins to try to locate the source of the problem. This effort usually entails a partial reconstruction of what transpired and how to make it appear again. Well-organized system development documentation helps in the search, but if it is not available, people who might know about how the system was developed in its current form must be found and engaged. This situation can further deteriorate if the attribution of responsibility for the bug or its adverse effects is in question. Thus, if the reconstruction is marked by uncertainty and frequent negotiations, participants may subsequently choose to work around the system aberration leaving it for future staff to rediscover, reconstruct (again), and attempt to rectify.

**Documentation:** Documentation is both the record and outcome of the preceding life cycle activities. Documentation represents the most tangible product of system development activities. Without it, the usefulness of the system is limited. However, the utility of the various life cycle documents is short unless effort is directed to continually update them. We more commonly hear more about (and experience) the inadequacy of available system documentation than of its superlative comprehensiveness. Standards and incentives for good documentation are few. Users need one kind of documentation, developers another, and maintainers possibly a third. If the system development effort was erratic or behind schedule, then documentation work may be put off. Documentation work is labor intensive and revealing of personal communication skills. System evolution continually makes obsolete available documentation unless countervailing support is provided. Further, system aberrations may not be documented since they may be used as evidence indicating a lack of competence by certain participants. In order to assure high-quality of the most visible, and in the long term, the most important products of system life cycling, development and use of system documentation must be planned, organized, staffed, controlled, coordinated and scheduled as the system must be. The system is its documentation.

**Use:** How software systems get used is not well understood from a SEPM viewpoint. Are well-engineered software systems easier to use? Who is to say? Apparently, system use is shaped by 1) the discretion a participant has over when

and for what he/she can use the systems, 2) how easy it is to learn how to use an unfamiliar system, 3) what kinds of mistakes or errors are likely to be encountered, 4) how easy it is to work around system limitations, and 5) how integrated is the system's fit with on-going work routines [26]. Each of these arrangements is articulated only after a period of hands-on use of the system. These conditions cannot be thoroughly predicted during initial system development. However, as new systems are cycled through participants' work, the work routines change and subsequently so must the system. The system also changes as staff turnover thereby requiring new staff to (re)negotiate the arrangements which shape their use of the system. Thus, circumstantial conditions in the work setting play a large role in determining the pace at which a software system is consumed and evolved.

*Evaluation and Maintenance:* Local participants regularly evaluate how well their systems work and how useful they are. As their experience with a system grows, so will the system's apparent inadequacy. In turn, participants will seek system enhancements, adaptations, repairs, or conversions as this occurs. Each of these maintenance activities entail a partial reenactment of system development. Maintenance is ongoing, incremental system development. As such, maintenance is also part of the process of innovation in computing [37]. The care and attention to detail by which maintenance work gets done shapes long-term system usability. However, many conditions counter an ideal practice of system maintenance: 1) users often have more requests for enhancements than can be realized by system maintainers; 2) poor quality of development documentation complicates the ease of figuring out where to make system alterations, 3) maintenance work often competes with new system development on specialists' agenda, 4) multiple system versions appear when maintenance activities are not coordinated or when unwanted alterations are resisted by users, 5) turnover of system development staff fragments local product knowledge; and 6) bureaucratic mechanisms such as change control boards create a new source of resistance that must be engaged (or bypassed) in order to keep the system well-integrated into ongoing work routines. As maintenance activities lag, users begin to either take on maintenance work in order to keep the system useful or they work around the system. Subsequently, as this arrangement becomes too demanding for users or as new technological alternatives appear, participants may let the system sink in order to establish the "need" to adopt a new system. This marks the termination of one system life cycle and the initiation of another.

Overall, there is a high degree of concurrency across activities occurring during a software system's life cycle. The life cycle is more circular than linear. But as we move to improve our ability to engineer this cycling, a growing array of resources must be committed, new forms or subdivisions of work emerge, and a more complex web of arrangements appears which must be managed.

#### V. UNDERLYING RELATIONSHIPS

From our starting assumptions and our analysis of historical events and present software life cycle conditions, five recurring patterns of social action in SEPM can be identified. These

five observations provide an emerging account of what drives the social costs, organizational impacts, and benefits of SEPM practices and thus, the more traditional budgetary costs [27].

1) *Macrostructural conditions influence local software practices:* The circumstances of local SEPM activities are shaped, not only by exigencies of day-to-day working an organization, but also the market forces impinging upon an organization, the production processes and products characteristic of the organization's industry segment, the regional labor market for software professionals, and the introduction of new software technologies diffusing throughout the computing world. How these broader arrangements constrain or facilitate local SEPM practices is not well understood. However, such arrangements are known to be a determining force that drives the cost of developing operating, and maintaining other complex system technologies [14].

2) *The dynamics of software innovation, use, and evolution are not widely understood:* We do not yet know how to systematically produce software technologies that can be readily adopted, assimilated, used, evolved, and managed in different settings with locally available resources and talents. Software systems are often developed with an initial emphasis on system performance, then on variety of application, standardization of costs, and later on maintenance (cf. [43]). Accordingly, much effort now goes to the processes of fitting, repackaging, and cycling software systems within the local computing infrastructure in order to keep systems useful [37]. These processes of software consumption need further investigation to determine if and how they are amenable to engineering rationalization.

3) *Organizational arrangements shape the effectiveness of local software practices:* Software engineering and project management practices are specialized, to local organizational production processes, constrained by historical and present circumstances, and motivated by narrow incentives and opportunities. Control over access to computing resources is central to productivity, while the ease of access to these resources shapes the complexity of software project tasks. As mainstream software engineering practices are adopted in an organization, a) a new division of labor and system understanding emerges, b) more specialized skills and staff are needed, c) more computing resources are needed, and d) more interdependent activities need to be managed.

4) *The outcomes of negotiation and articulation work determines how software will be produced and consumed:* Participants negotiate a complex array of resources in articulating the course of their software work. The outcomes of these negotiations determine a) the growth and manageability of the local computing infrastructure, b) what affects the costs, organizational impacts, and benefits of local SEPM practices as well as c) how they will be distributed among participants. Therefore, the more frequently participants negotiate and articulate local software life cycle practices, the more project management will be distributed among participants.

5) *Perceived cost and benefit drivers are determined by the separability of the unit of analysis:* Should the focus of attention be some software development technology or how people will organize to work with that technology? What you see depends on how you look. Software engineering

and project management work takes place within a) dense web of social arrangements, b) evolving work patterns, c) uncertain conditions to choose between, and d) the proliferation of new computing technologies. Perceiving what diminishes or increases available supplies to computing resources is influenced by how we decompose the complexity of circumstantial computing arrangements. Similarly, one man's savings may be another's costs. That is, depending on how resource transfers are accounted and managed, savings may be internalized within one set of arrangements while costs are externalized onto others or into the future. In short, there is much work yet to be done in developing microeconomic and macroeconomic theories of software production and consumption as well as how such theories relate to local SEPM practices.

## VI. FUTURE DIRECTIONS: WHERE TO GO AND WHAT TO DO?

### A. Where Are We Headed?

The principal technical activity of software engineering is moving toward something akin to "software redevelopment." Software redevelopment means taking an existing software description (e.g., as expressed in a programming or very high-level language) and transforming it into an efficient, easier-to-maintain realization portable across local computing environments. This redevelopment technology would ideally be applicable to redeveloping both 1) rapidly assembled system prototypes into production-quality systems, and 2) old procrustean software developed 3-20 years ago still in use and embedded in ongoing organization routines but increasingly difficult to maintain. In addition, redevelopment technology could be used to help structure the production of new system components intended for frequent reuse. Many researchers are also forecasting increasing wide-spread adoption of software engineering environments operating on networks of personal computing workstations targeted to support application-specific processing. Ultimately, the future of software engineering and project management will be an outgrowth of historical trends, current practices, and local circumstances.

Our understanding of the complex web of social arrangements that situate local SEPM practices will likely lag behind our ability to develop new software technologies. This points to yet another dilemma: will present SEPM strategies be appropriate as new technology advances? Based on the preceding analysis of historical trends and current practices, software management strategies rooted in automated or bureaucratic mechanisms will become less workable in current form and thus require revision. That is, SEPM mechanisms have life cycles too. This suggests that a crucial factor in the productive, long-term deployment of new software production technologies is the ease with which these mechanisms can be developed, packaged, fit, and cycled into local SEPM practices [37].

At present, there is no formal theory of software production and consumption that can serve as a guide for practical action. Although there are bits and pieces of substantive theory on which to build (as described above), we need a more extensive empirical base of the interplay of social,

economic, and technological, arrangements brought to bear in life cycling software. Most of the social arrangements for SEPM examined above are not addressed well through new technological and bureaucratic mechanisms that emerge from idiosyncratic circumstances. These circumstances play a major theoretical and practical role, yet they usually escape careful scrutiny. So, what do we do? What strategies should software engineers and project managers pursue?

From what we know so far, the following provides a partial set of strategies to pursue.

### B. What to Do

1) *Identify the web of arrangements that surround local software production and consumption:* The suggestion here is to get a broad picture of the local setting for computing work in terms of available resources and the life cycle activities that process them. The resources of interest are those that are subject to contention or negotiation among key participants during life cycle activities. These resources will usually appear packaged together to include computing hardware, systems and application software, time, money/budgets, organizational units, staff, staff skills, management attention, information control, and prevailing beliefs about local computing arrangements. Accordingly, the "costs" of local SEPM practices will represent new resource commitments or expenditures, "benefits" will represent new resource supplies, while "organizational impacts" will represent shifts in the patterns of resource allocation or distribution. In other words, depending where stand, resources shifting toward you are benefits, while those shifting away are costs, assuming all other things are unchanged. Determining whether the introduction of a new software tool or technique will have a more of a beneficial or costly impact will then be determined by where you are standing, and whether the new resource arrangement is more or less desirable than the existing one by the participants in that position. Thus, as participants move into different positions throughout a system's life cycle, then software production will become more costly and problematic as participants find themselves working in adversely impacted situations. The same relationship holds for software consumption.

2) *Adopt/establish routines for producing software:* Software production can be ordered through use of development methodologies or standard operating procedures (SOP's). The purpose of such standardization is to avoid certain kinds of conflicts, establish lines of authority for resolving these conflicts, and routinize work procedures. SOP's for production of software documentation are necessary, although insufficient for assuring high-quality. Also, SOP's to include the participation of target users throughout software development and maintenance should help. Since SOP's characterize situations that can be articulated and rationalized, creation of SOP's should be carefully formulated to minimize built-in conflict situations. SOP's are never comprehensive and their utility is subject to change as unexpected circumstances arise. Clever (or covert) ways to work around the SOP's will emerge and should be expected. Thus, when work-arounds of particular SOP's become more frequent or patterned, the SOP should be reformulated and rearticulated. As such, system

development methodologies, as one family of SOP's, will also have a life cycle. Needless to say, planning, organizing, staffing, controlling, scheduling, directing, cost estimating, and life cycling software projects will expectedly be prime activities to be (at least partially) converted into SOP's.

3) *Identify objectives for the software production process:* To begin, articulate the flow of resources into the lattice of production and consumption activities. What is important here is to outline local resource network in order to identify available "upstream" supplies, their distribution, allocation and access policies, bottlenecks, and technology packages as well as how they are brought together into "downstream" products. Also characterize the broader arrangement, that affect the availability of resources on conflict-laden or otherwise critical paths. Second, outline the cluster of SOP's that transform available resources into products. These SOP's should cover the activities occurring during the software life cycle as practiced in the focal setting. Finally, map the resource network onto the clusters of SOP's to outline the flow of products through the production process within local productive units. Accordingly, objectives such as increased product reliability, reduced development costs, or increased user orientation can be understood as policies that direct resource flow, formulate SOP's, or alter broader social arrangements.

4) *Design project organization to facilitate commitment:* Everyone will be responsible for managing some portion of overall work activities. Since project managers will be responsible for coordinating work and resources within the local computing infrastructure, they need to know about bottlenecks and other troublesome conditions. Maintaining a high-level of software production requires the commitment of staff and resources to achieve it. Continuity of commitment is more central than control, since control is distributed and more subject to contention. Maintaining staff commitment requires regularly assessing the conditions that bind their commitment to work: desired resource availability, local (dis)incentives, and career opportunities. Such an assessment emerges when staff participate in deciding how to realize project objectives. The regularity of assessment depends on the perceived stability or uncertainty of local SEPM conditions. Unexpected circumstances will always emerge and give rise to destabilizing conditions. However, strong commitment will often provide staff members idiosyncratic motivation to accommodate local contingencies until the prevailing order is reestablished, unless their commitment is sufficiently weakened.<sup>4</sup> But if their commitment to project objectives is strong, so that their perceived investment (or stake) in project activities is clear, then they can build on their investment by discovering new ways to perform their work.

5) *Develop new software technology as a package:* Every software technology (or system) assumes some configuration of hardware, existing software base, documentation, time, money, skills, organizational units, management attention, and other resources for its productive use. This package

<sup>4</sup> The building of strong commitment for some ("signing up"), and the weakening of commitment for others ("burning out") is a key element in what gives a new system its "soul" [19].

of resources outlines a set of requirements that must be met by participants working within the local computing infrastructure. The package must fit into the local setting. As such, users need to know what resource requirements are built into a new technology in order to assess both the costs and ease with which it can be fit into existing computing arrangements. However, as a new technology is fitted and assimilated into ongoing organizational routines, the local computing infrastructure will be altered to reflect its repackaging. Historical trends in software engineering indicate that this repackaging is done to make the local production process more productive and routine. However, these trends also indicate a greater division and specialization of labor among participants as well as an increase in the number of resources will need to be coordinated. Accordingly, an important cost of using new software technologies to make the local production process more productive is increased demands for attention to detail and routine. This is a form of management that individual participants must increasingly perform. Thus, in developing a new software systems, tools, or engineering methodology, a strategy for managing its life cycle must explicitly be built into its package.

6) *Finally, build empirical databases for comparative analysis:* We lack empirically grounded, theoretical understandings of software production and consumption. Pursuing the preceding strategies will undoubtedly reveal a larger set of problems or troublesome conditions in SEPM. This is not bad. Rather, each strategy is an informing source of feedback on the accuracy of a current understanding. A systematic analysis of an organization's history and present circumstances will suggest possible future arrangements as well as the pace at which they can be achieved. However, the cost of such an analysis grows as the scope and generalizability of subsequent findings is enlarged. Learning what to do in unfamiliar or troublesome situations comes from preparation, practical experience, and a comparative framework to link them. The payoff here is long term. While this strategy was suggested more than 10 years ago by Boehm [3], most readily available archival databases of real-world SEPM case studies (such as this journal) are sparsely populated. Perhaps it is time that more research should be funded to increase the population of studies for subsequent comparative analysis.

## VII. CONCLUSIONS

Software engineering and project management are performed within a dense web of technological, economic and social arrangements. The number, interdependence, and specialization of the particular arrangements reflect their complexity. The causal patterns and strategies for what to manage described above characterize the current model of social action for SEPM within these settings.

In presenting this material, choices were made on what topics to include and what topics to put off for discussion elsewhere. Important topics for the social analysis of SEPM not covered here include 1) developing or evaluating research methodology, 2) the (in)adequacy of analytical tools and notations for this kind of analysis, and 3) the difficulty of making theoretically persuasive explanations or arguments for practical action. However, drawing from the topics covered

in this paper, social analysis emerges as one of the most revealing approaches to software engineering and project management.

A basic question that motivated this paper was to what extent do social arrangements impinge upon local software engineering and project management practices. The answer to this question came through an analysis of historical and contemporary studies. The analysis shows that a complex web of social conditions significantly determines the life cycle of local software systems and how easily such systems can be managed. Further, the extent to which an analyst assumes that local social arrangements can be separated from an analysis of the software technology, system life cycle, life cycle costs, and their management determines the scope of the analyst's observations as well as the unexplained dilemmas that remain.

As Boehm [6] observed, in order to engineer software systems that are useful to people, "... concerns for the social implications of computer systems are part of the software engineer's job, and techniques for dealing with these concerns must be built into the software engineer's practical methodology, rather than treated as a separate topic isolated from our day to day practice" (emphasis added). As argued in this paper, we need to focus further attention not only on the social implications of computing systems, but also how the complex web of social arrangements shapes the production and consumption of software systems, the local software engineering practices, the distribution of costs and benefits, the appropriateness of new software technologies, and the ease with which these can be managed. This paper marks a step in that direction.

#### ACKNOWLEDGMENT

The author would like to acknowledge the collegial support and commentary shared with L. Gasser, E. Gerson, E. Horowitz, J. L. King, R. Kling, A. Strauss, and others in discussing the material presented here. Also, he would like to thank A. Pyster and R. Thayer, who played a key role in providing both the opportunity and encouragement to develop this paper.

#### REFERENCES

- [1] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 2, no. 1, pp. 56-78, 1972.
- [2] L. Belady, "Large software systems," T. J. Watson Res. Cen., Yorktown Heights, NY, IBM Rep. RC6466, 1978.
- [3] B. Boehm, "Software and its impacts: A quantitative assessment," *Datamation*, vol. 19, pp. 48-59, May 1972.
- [4] —, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, no. 12, pp. 1226-1241, 1976.
- [5] —, "Software engineering: R&D trends and defense needs," in *Research Directions for Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979, pp. 44-86.
- [6] —, "Software engineering: As it is," in *Proc. 4th Int. Conf. Software Eng.*, IEEE Comput. Soc., 1979, pp. 11-21.
- [7] F. P. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
- [8] J. Huxton, P. Naur, and B. Randell, *Software Engineering: Concepts and Techniques*. Petroselli Books, 1976.
- [9] F. J. Corbato and C. T. Clingen, "A managerial view of the multics system development," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: MIT Press, 1979, pp. 139-158.
- [10] J. Danziger, "The skill bureaucracy and interorganizational control," *Sociol. of Work and Occupations*, vol. 6, pp. 204-226, 1979.
- [11] B. C. DeRoze and T. H. Nyman, "The software life cycle—A management and technological challenge in the Department of Defense," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 309-313, 1978.
- [12] J. Distaso, "Software management—A survey of practice in 1980," *Proc. IEEE*, vol. 68, pp. 1103-1119, 1980.
- [13] R. R. Everett, C. A. Zraket, and H. D. Benington, "SAGE—A data processing system for air defense," in *Proc. East Joint Comput. Conf.*, AFIPS Press, 1957, pp. 148-155.
- [14] J. S. Gansler, *The Defense Industry*. Cambridge, MA: MIT Press, 1980.
- [15] J. Guenther, *Management Methodology for Software Product Engineering*. New York: Wiley-Interscience, 1978.
- [16] W. Husier, "Pitfalls and safeguards in real-time digital systems with emphasis on programming," *IRE Eng. Management*, vol. EM-8, pp. 99-115, June 1961.
- [17] P. G. W. Keen, "Information systems and organizational change," *Commun. Ass. Comput. Mach.*, vol. 24, no. 1, pp. 24-33, 1981.
- [18] P. G. W. Keen and E. Gerson, "The politics of software system design," *Datamation*, vol. 24, pp. 80-84, Nov. 1977.
- [19] T. Kidder, *The Soul of a New Machine*. Atlantic Monthly Press, 1981.
- [20] J. L. King and E. T. Schrems, "The costs and benefits of information systems development and operation," *ACM Comput. Surveys*, vol. 10, no. 1, pp. 19-34, 1978.
- [21] R. Kling, "The organizational context of user-centered software design," *MIS Quart.*, vol. 1, no. 3, pp. 41-52, 1977.
- [22] —, "Social analyses of computing: Theoretical perspectives in recent empirical research," *ACM Comput. Surveys*, vol. 12, no. 1, pp. 61-103, 1980.
- [23] R. Kling and E. Gerson, "Patterns of segmentation and intersection in the computing world," *Symbolic Interaction*, vol. 1, no. 2, pp. 24-43, 1978.
- [24] R. Kling and W. Scacchi, "The DoD common high order programming language (ADA): What will the impacts be?" *SIGPLAN Notices*, vol. 14, no. 2, pp. 29-43, 1979.
- [25] —, "Recurrent dilemmas of computer systems use in complex organizations," in *Proc. 1979 Nat. Comput. Conf.*, AFIPS Press, vol. 48, 1979, pp. 107-116.
- [26] —, "Computing as social action: The social dynamics of computing in complex organization," *Advances in Computers*, vol. 19, pp. 250-327, 1980.
- [27] —, "The web of computing: Computing technology as social organization," *Advances in Computers*, vol. 21, pp. 3-78, 1982.
- [28] D. W. Kosy, "Air Force command and control information processing in the 1980's: Trends in software technology," Rand Corp., Santa Monica, CA, R-1012-PR, 1974.
- [29] P. Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States*. New York: Springer-Verlag, 1977.
- [30] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Reading, MA: Addison-Wesley, 1980.
- [31] H. C. Lucas, *Why Information Systems Fail*. New York: Columbia Univ. Press, 1974.
- [32] H. Mintzberg, *The Nature of Managerial Work*, 1973.
- [33] J. Palme, "How I fought with hardware and software and succeeded," *Software—Practice and Experience*, vol. 8, no. 1, pp. 77-83, 1978.
- [34] A. M. Pettigrew, "Information control as a power resource," *Sociology*, vol. 6, pp. 179-205, 1972.
- [35] R. E. Quinnan, "The management of software engineering part V: Software engineering management practices," *IBM Syst. J.*, vol. 19, no. 4, pp. 466-477, 1980.
- [36] D. J. Reifer, *Software Management: A Tutorial*, 2nd ed. IEEE Comput. Soc. Press, 1982.
- [37] W. Scacchi, "The process of innovation in computing: A study of the social dynamics of computing," Ph.D. dissertation, Dep. Inform. Comput. Sci., Univ. California, Irvine, 1981.
- [38] W. Scacchi, L. Gasser, and E. M. Gerson, "Problems and strategies for organizing computer-aided design work," *Proc. IEEE ICCAD'83*, 1983.

- [39] C. H. Sequin, "Managing VLSI complexity: An outlook," *Proc. IEEE*, vol. 71, no. 1, pp. 149-166, 1983.
- [40] H. A. Simon, *Administrative Behavior*. New York: Macmillan, 1947.
- [41] N. Sondheim, "On the fate of software enhancement," in *Proc. 1978 Nat. Comput. Conf.*, AFIPS Press, vol. 47, 1978, pp. 1158-1163.
- [42] R. Thayer, A. Pyster, and R. Wood, "Major issues in software engineering project management," *IEEE Trans. Software Eng.*, vol. SE-7, July 1981.
- [43] J. M. Utterback and W. J. Abernathy, "A dynamic model of process and product innovation," *Omega*, vol. 3, no. 6, pp. 337-356, 1975.
- [44] G. Weinberg, *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.
- [45] R. W. Wolverson, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, no. 6, pp. 615-636, 1974.



Walt Scacchi (S'77-M'80) received the B.S. degree in computer science and the B.A. degree in mathematics from California State University, Fullerton, in 1974 and the Ph.D. degree from the University of California, Irvine, in 1981.

He is currently an Assistant Professor in the Department of Computer Science, University of Southern California, Los Angeles. His major research interests are in understanding the process of innovation in computing. He is actively engaged in research in software engineering, artificial intelligence, and the management of computing.

Dr. Scacchi is a member of IFIP Working Group 9.2 (Social Accountability in Computing), the Association for Computing Machinery, the Society for the History of Technology, the American Association for Artificial Intelligence, and the Society for the Study of Symbolic Interaction.

## Making Software Visible, Operational, and Maintainable in a Small Project Environment

WILLIAM BRYAN AND STANLEY SIEGEL

**Abstract**—Practical suggestions are presented for effectively managing software development in small-project environments (i.e., no more than several million dollars per year). The suggestions are based on an approach to product development using a product assurance group that is independent from the development group. Within this check-and-balance management/development/product assurance structure, a design review process is described that effects an orderly transition from customer needs statement to software code. The testing activity that follows this process is then explained. Finally, the activities of a change control body (called a configuration control board) and supporting functions geared to maintaining delivered software are described. The suggested software management practices result from the experience of a small (approximately 100 employees) software engineering company that develops and maintains computer systems supporting real-time interactive commercial, industrial, and military applications.

**Index Terms**—Configuration control board, design review, product assurance, project management, testing.

Manuscript received November 5, 1981; revised October 1, 1982.  
The authors are with CTEC, Inc., 6862 Elm Street, McLean, VA 22101.

### I. INTRODUCTION

In *The Mythical Man-Month* [1], Brooks asks, "Why is programming fun?" He then offers five reasons, the fifth one being the following (page 7):

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.

The tractability of the software medium that Brooks refers to is, we maintain, the basic challenge to software engineering project management. The purpose of this paper is to offer practical suggestions for taking on this challenge with a reasonable degree of confidence.

Our suggestions are *practical* because they are:

- derived from techniques successfully applied in the real world;



- applicable to the large range of projects that involve from a handful up to 30 or so people, or stated in other terms, whose budgets span from tens of thousands to millions of dollars;

- based on good business sense and therefore saleable to corporate management;

- grounded in common sense and therefore adaptable to other organizations. (We believe a philosophy of "try it, you'll like it" that appeals to basic reason and is tempered with patience—as opposed to a "do it because we tell you" philosophy—is an effective way to contain poetic license).

Our suggestions are geared to:

- *making software visible*, or transforming software into something that management and others can see (our notion of *software* encompasses specification documentation as well as its resultant computer code—see [2, ch. 1]),

- *making software operational*, or producing software that performs according to stated customer needs,

- *making software maintainable*, or being able to modify software in response to revised customer needs or identify discrepancies with respect to these needs.

This paper does *not* relate a case study of one company's successful application of software engineering project management techniques to *one* particular project. Rather, it reflects a corporate attitude toward performing software engineering project management on *any* business endeavor. For this reason, we believe that our suggestions may be, at least in part, beneficial to others. We do not claim that these suggestions are all-encompassing (we have project management problems, too); nor do we claim that these suggestions are directly applicable to a corporate environment other than ours. [The dogmatic application of a successful technique in our company to the problems of another company—even one of similar size and structure—may fail for a variety of reasons, such as corporate politics and/or policies; for example, some company executives are simply willing to live with the pain of some problems (i.e., maintain the status quo) rather than subject themselves to the trauma of change that potential solutions might invite.]

This paper addresses the following topics:

- 1) An Independent Product Assurance Group (Section II).
- 2) Transitioning from a Statement of Customer Needs to Software Code—The Software Development and Change Control Process (Section III).
- 3) Determining that Operating Software Code is Consistent with Customer Needs—The Testing Cycle (Section IV).
- 4) Keeping the Customer Satisfied—The Configuration Control Board (CCB) and the Maintenance Process (Section V).

Section VI summarizes the key points. For easy reference, this summary is in the form of suggestions for organizing and managing a software project.

## II. AN INDEPENDENT PRODUCT ASSURANCE GROUP

We work for a firm, CTEC, Inc., that started as a one-person management consulting company in March 1974. In 1976, a client asked the company to pick up the pieces from a failing software-development effort. Since that time, we have been in the business of developing, fielding, and

maintaining operational systems with software content. The size of our software projects ranges from tens of thousands to hundreds of thousands of lines of programming language (higher order and/or assembler) source code; the types of software that we develop and maintain support real-time interactive commercial, industrial, and military applications. The company has grown steadily and has achieved a business base of just under \$8 million. In 1978, the company reorganized into a matrix-managed organization (described later in this section). A software development process (see Section III) meshing with this organization has gradually become standardized. The process has even caused the extension of the matrix-managed organization to a third dimension: product assurance. The forces of this third dimension stabilize the software development process and manifest themselves primarily as peer reviews (see Section III). They impress visibility and traceability on the software development process.

Before looking more closely at this three-dimensional approach to matrix management, it is instructive to briefly consider the disciplines needed for making visible, operational, and maintainable software. Required is the interplay of three groups of disciplines—development, product assurance, and management—as illustrated in Fig. 1 and described in the following paragraphs.

- The *development discipline* shoulders the responsibility for creating the software during its various life cycle stages and doing what must be done to get the product into the hands of the customer, e.g., analysis, design, coding, and training.
- The *product assurance discipline* provides management with checks and balances with respect to the developer's activities. As Fig. 2 illustrates, these checks and balances help assure that product integrity is attained, and, ultimately, that the customer is satisfied. By product integrity we mean a product:

- that fulfills customer needs
- that can be easily and completely traced through its life cycle
- that meets specified performance criteria
- whose cost expectations were met
- whose delivery expectations were met (see [2, pp. 58-59]).

Fig. 2 also indicates that we view product assurance as the interplay of the functions of quality assurance (QA), configuration management (CM), verification and validation (V&V), and test and evaluation (T&E). The identifiers QA, CM, V&V, and T&E mean different things to different people, as evidenced by the wide variation of meaning associated with these terms in the literature (e.g., one person's QA is another person's T&E). For us, the world of product assurance is divided into the four processes shown in the far left of Fig. 2. We believe that these processes are necessary to help assure product integrity. We assign these processes the function labels shown in Fig. 2 to provide some linkage with extant, albeit nonuniform, terminology. The boundaries between these four processes are not distinct; their domains overlap (e.g., T&E can be viewed as a form of QA in which the "stand-

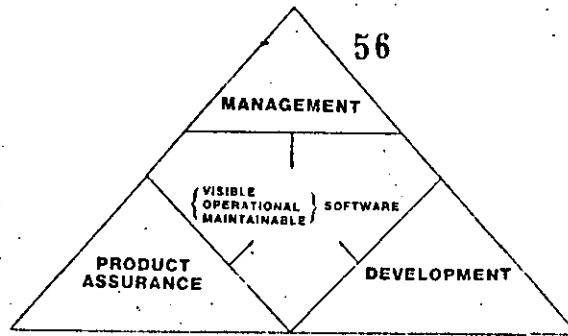


Fig. 1. Requisite disciplines for making visible, operational, and maintainable software.

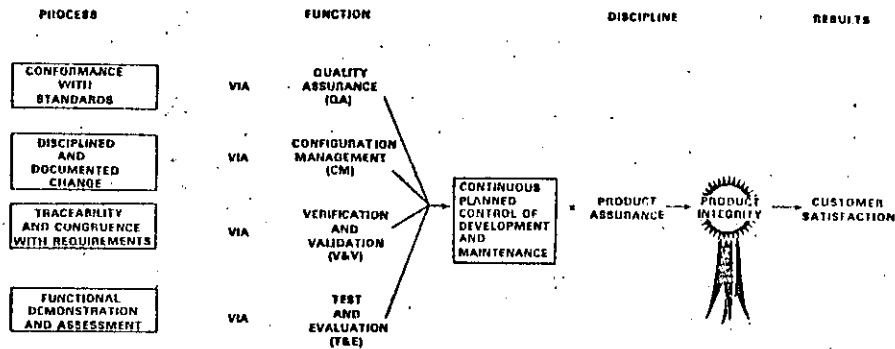


Fig. 2. Product assurance: Its functions and the results of its application.

ard" is a test plan or procedure against which the coded form of the software operating on hosting hardware is being compared). The function labels and process overlaps are not important here. What is fundamental is the integrated performance of all these processes.

• The *management discipline*, which can be divided into project management and general management, provides direction to development and product assurance activities to effect synergism. Project management provides this direction generally at the level of day-to-day activity associated with product development. General management provides this direction generally at the level above a particular project organization. Typically, this direction concentrates on sorting things out with respect to two or more projects that may be competing for corporate resources.

Fig. 3 depicts an organizational structure that we have used to develop and maintain software systems. This figure is a specific implementation of the philosophy represented in Fig. 1, where each discipline is depicted as an axis in a three-dimensional space. Along the "development axis" are three functional departments: 1) Systems Analysis and Design, 2) Systems Engineering, and 3) Software Engineering. The Systems Analysis and Design Department is concerned with defining customer requirements, developing solution approaches (in the big-picture sense, such as architectural level tradeoff studies), and designing algorithms for specific mathematical

problems associated with a system under development. This department is staffed with individuals trained in operations research, systems analysis, and computer science. The Systems Engineering Department is concerned with performing a top-level hardware/software functional allocation (i.e., specifying which system functions are to be carried out by hardware and which are to be carried out by software). This department is staffed with individuals trained in human factors analysis, hardware engineering, and communications engineering. The Software Engineering Department is concerned with detailed software design, software coding (programming), and program debugging. This department is staffed with individuals trained in computer program design and conversant with one or more programming languages.

Along the "management axis" in Fig. 3, there are project management offices that report to corporate management. A project manager is appointed for each project or set of related projects. Each project manager draws upon the resources of the three departments just described. However, the project manager does not assume authority over these resources. The staff continue to report to their respective department managers. [The row/column (i.e., matrix-like) organizational setup shown in Fig. 3 gives rise to the terminology *matrix management*.]

Along the "product assurance axis" in Fig. 3 is the Product Assurance Department. This department plays the role of the

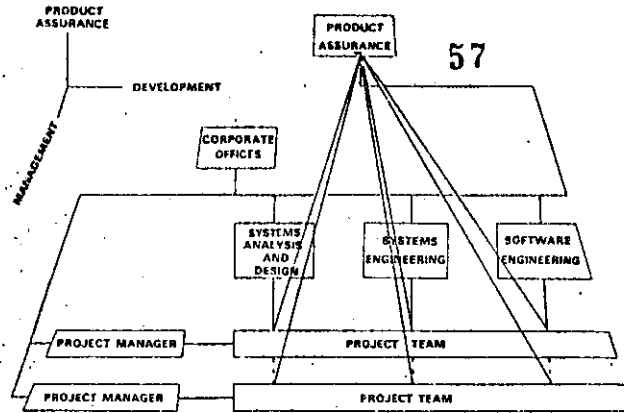


Fig. 3. A three-dimensional organizational structure for making visible, operational, and maintainable software—a "plane" of matrix management augmented by a "third dimension" of product assurance.

devil's advocate, providing each project manager (and corporate management) with an avenue to gain visibility into project progress other than through the three functional departments. Typically, these departments and the Product Assurance Department perceive project progress from different viewpoints—the three functional departments perhaps more optimistically and the Product Assurance Department perhaps more pessimistically. This department thus provides management with a potentially contrasting view of project progress so that management has the opportunity to make more intelligent decisions. To ensure its effectiveness, the Product Assurance Department is separated organizationally from the functional departments and the project managers. Thus, the department's objectivity is maintained which, in turn, maintains its effectiveness. This posture provides corporate management with an added measure of assurance that projects are proceeding on schedule, within budget, in a traceable manner, and in accordance with customer requirements and performance criteria (and, if projects are not proceeding in this manner, this department offers corporate management an opportunity to find out why). For example, through the configuration control board mechanism described in the following two sections, a visible trace of project activities is compiled and maintained by the Product Assurance Department, providing management with an essential input for making intelligent decisions regarding subsequent project evolution.

### III. TRANSITIONING FROM A STATEMENT OF CUSTOMER NEEDS TO SOFTWARE CODE—THE SOFTWARE DEVELOPMENT AND CHANGE CONTROL PROCESS

This section describes our software development and change control process in terms of the three-dimensional organizational structure portrayed in Fig. 3. This process aims at making software development and change control visible—and thus manageable—activities. The result is a software product that is operational (i.e., meets customer needs) and is maintainable once it is fielded.

Fig. 4 portrays the flow of our software development and change control process. The cyclic flow depicted in the figure is based upon the major release approach. This approach consists of periodically (i.e., approximately every 6 to 12 months) incorporating a group of enhancements into an existing operational baseline to create the subsequent operational baseline. This group of enhancements, when incorporated into an existing operational baseline and deployed, constitutes a *major release*. A major release is thus a controlled way of upgrading a deployed system in roughly uniform increments (in contrast to an approach that upgrades a deployed system each time a change is approved—regardless of the magnitude of the change). The primary advantage of the major release approach is that it permits changes to be more effectively integrated with one another and with capabilities in the current operational baseline. The primary disadvantage of the major release approach is that the customer generally has to tolerate system weaknesses and problems for a longer period of time.

The following is a walk-through of Fig. 4 (the numbers in circles are keyed to the paragraph numbers given below).

1) Customer requirements (as articulated in a contract) are translated into software code via a sequence of progressively more detailed specification steps with each step in the sequence formalized by a design review (coding does not begin until all design reviews are completed). This design review cycle (described in more detail in [3]) generally involves the development and review of the following three specification documents describing the enhancement to be incorporated into the next major release.

a) Preliminary Design Document (PDD), which translates the customer requirements into a functional approach to the solution. The document contains data flow descriptions and a top-level system concept incorporating the enhancement. The primary purpose of the PDD is to provide sufficient documentation for a more detailed design of the enhancement after top-level consideration of alternatives

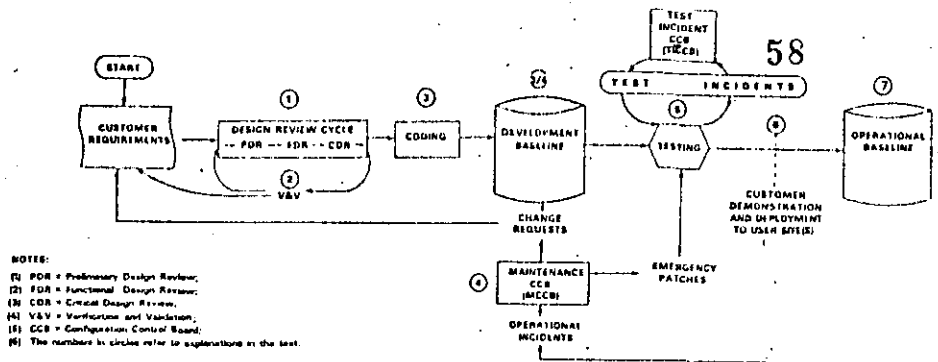


Fig. 4. Overview of CTEC's software development and change control process.

for implementing the enhancement. The PDD is generally prepared by the Systems Analysis and Design Department. It is presented for review, modification, and/or approval at a Preliminary Design Review.

b) Functional Design Document (FDD), which focuses on allocating the functions identified in the PDD to hardware and software, and explaining the functions and implications of the design. The FDD typically contains the syntax for command strings and formats for menus that comprise the interface between the user and the system. The FDD is generally prepared by the Systems Engineering Department. It is presented for review, modification, and/or approval at a Functional Design Review.

c) Critical Design Document (CDD), which describes the design of specific software modules needed to implement the commands, menus, and other functions described in the FDD. The CDD typically contains logic diagrams (e.g., Warnier-Orr diagrams) which are then transformed into software code (i.e., programming language statements) during the coding stage shown in Fig. 4. The CDD is generally prepared by the Software Engineering Department. It is presented for review, modification, and/or approval at a Critical Design Review.

Depending on the level of detail in the customer requirements, the PDD and/or the FDD may be omitted from the design review cycle. For example, a contract with a customer may already contain as an appendix a document comparable to an FDD developed, say, by another contractor. In such cases, a design review may be held at the outset of a project to determine if any changes need to be made to the customer-provided FDD (or PDD). The customer generally participates in the Functional Design Review and Critical Design Review and formally approves the associated design documentation. If necessary, the customer requests changes to this documentation (which may then be submitted for customer approval at a subsequent design review). The Preliminary Design Review is generally not attended by the customer, since this review is a formal brainstorming session on how to design for a customer requirement. The PDD may be made available to the customer as a progress report.

2) Throughout the design review cycle, the Product Assurance Department performs verification and validation (V&V)

of the design documentation. This department verifies that each document follows logically from its predecessor and validates that each document is consistent with customer requirements. As is explained in Section IV, the Product Assurance Department is also responsible for developing and executing procedures to test the development baseline (Fig. 2). The V&V activity of this department provides the department with early inputs for these procedures. (See [4] for a discussion of verification and validation for software design documentation and for all other software products developed during the life cycle.)

3) Once the CDD is approved, the Software Engineering Department prepares code in accordance with the CDD and FDD. This code is then integrated with the code from the existing operational baseline to form a development baseline.

4) Also added to the development baseline is code from change requests approved by a Maintenance Configuration Control Board (MCCB). The MCCB is a change control body that functions throughout the period between major releases (and, in particular, throughout the design review cycle). The MCCB is described in Section V. (See [2] and [5] for a more detailed discussion of the concepts of change control and CCB as presented in this paper.)

5) Once the development baseline is constructed, it is turned over to the Product Assurance Department for testing. The testing cycle and the associated activities of the Test Incident Configuration Control Board (TICCB) are described in Section IV.

6) At the end of the testing cycle, the operation of the development baseline is demonstrated to the customer. Generally, this operation is demonstrated at our facilities and then at the user sites. This demonstration consists of executing the test procedures used during the testing cycle referred to in 5 above. This demonstration also consists of excursions from these test procedures that the customer may select to satisfy himself that the software code is performing in accordance with specifications. Sometimes, the customer may choose to have the demonstration at the user sites conducted by an agent other than our company, using either our test procedures or other test procedures.

7) If the customer is satisfied with the onsite demonstra-

tion, the development baseline replaces the existing operational baseline and becomes the new operational baseline. The customer may accept the development baseline as the new operational baseline even if all test incidents have not been resolved. These outstanding test incidents are then considered operational incidents and submitted to the MCCB for resolution.

The preceding discussion completes the walk-through of Fig. 4. The next section focuses on the testing cycle and the activities of the TICCB shown in the figure.

#### IV. DETERMINING THAT OPERATING SOFTWARE CODE IS CONSISTENT WITH CUSTOMER NEEDS—THE TESTING CYCLE

The testing cycle begins with a turnover of the development baseline by the Software Engineering Department to the Product Assurance Department.<sup>1</sup> This turnover is formalized by a memorandum from the Software Engineering Department to the Product Assurance Department listing all the change requests and software modules that are being handed over. Before the conclusion of the turnover meeting, a date is agreed upon for returning the development baseline (and test incidents) to the Software Engineering Department. (This initial turnover meeting can be regarded as the first TICCB meeting in the testing cycle depicted in Fig. 4.) The Software Engineering Department also turns over the media containing the development baseline software code. The Product Assurance Department makes a copy of this code and places the copy under configuration control. This copy becomes the reference point against which subsequent changes to the development baseline are made as a result of test incidents.

With the establishment of a configuration-controlled development baseline, the Product Assurance Department begins executing its test procedures. The test procedures document typically consists of several hundred pages and generally grows with each major release. It contains tests developed for change requests and patches, tests developed for the new capabilities being incorporated as a result of the most recent design review activity, and tests developed for capabilities introduced in previous releases. Satisfactory execution of these procedures thus provides a level of confidence that 1) the new capabilities are performing in accordance with the specifications set forth in FDD's and CDD's, 2) incorporating these new capabilities has not introduced any problems in capabilities that were part of previous releases, and 3) correcting old problems has not introduced new problems or reintroduced other old problems previously fixed.

Each set of test procedures addressing a functional area is introduced by a brief description of the capabilities included in that functional area and the testing approach taken. The step-by-step test procedures are then presented using the

<sup>1</sup> It should be noted that the Software Engineering Department generally performs some testing on the development baseline before this turnover. This testing focuses on ensuring that source compilation and assembly errors are not present and that all necessary code is present. This testing also generally includes checking the operation of individual modules to ensure, for example, that inputs are properly accepted and outputs are properly generated.

five-column format shown in Fig. 5. This five-column format is particularly suited to the testing of interactive systems, where the user interface is via a keyboard input and the system response is via a display of information on one or more screens. The definition of each column is as follows.

1) Column 1, *Step*, provides an identifying number for each step in the test procedure.

2) Column 2, *Operator Action*, defines the action taken by the tester to perform a particular step of the test procedure.

3) Column 3, *Purpose*, explains why a particular step is being executed or a capability is being exercised.

4) Column 4, *Expected Results*, describes the system response or other result that is expected upon completion of the action described in the *Operator Action* column. The information in this column comes from FDD and/or CDD material, or modifications to this material as a result of MCCB-approved change requests (see Section V).

5) Column 5, *Comments*, contains information that may be helpful to the tester. Examples include a description of the rationale behind a particular sequence of test steps, boundary value information for the functions being tested, or commentary on possible problems or critical areas. This column may also contain references to particular change requests approved by the MCCB. These references provide a means for tracing specific test results back to software specifications which, in turn, provide a means for tracing back to customer requirements. In this manner, the customer can be shown during demonstration testing that the capabilities he asked for are indeed embodied in the executing software code.

As the test procedures are executed, the tester may notice a discrepancy between the results specified in the Expected Results column and the actual response of the system. When such a discrepancy occurs, the tester fills out the upper portion of the Test Incident Report (TIR) form shown in Fig. 6. The tester also attempts to recreate the discrepancy by re-executing the pertinent test steps. If he is successful, he checks YES on the TIR form; otherwise he checks NO. The tester may also attach additional information to the TIR form, such as a printer output or a hard copy of information appearing on a display screen at the time the discrepancy was encountered.

When all the test procedures have been executed, a TICCB meeting, which was scheduled at the first turnover CCB meeting, is convened. At this meeting, the Product Assurance Department formally returns the development baseline to the Software Engineering Department. The TIR's generated during the execution of the test procedures are discussed to clarify possible misunderstandings about the problems described on the TIR forms. Following the meeting, the Product Assurance Department writes a memorandum such as that shown in Fig. 7 that summarizes what happened at the meeting. This memorandum gives visibility to the testing process because it accomplishes the following:

1) Formally establishes that the development baseline is now back in the hands of the Software Engineering Department.

2) Gives pointers<sup>2</sup> to the problems discovered during

<sup>2</sup> That is, the TIR numbers (in this case, #1 to #105).

**60**

<b>TEST XX.Y</b>	<b>OBJECTIVE:</b> This area contains a statement that defines the objective of Text XX.Y.			
	<b>TITLE:</b> This line contains the long title of the test procedures.			
	<b>NOTES:</b> This area provides general notes as required to execute the test procedures within a section.			
STEP	OPERATOR ACTION	PURPOSE	EXPECTED RESULTS	COMMENTS
N	Describes the actions taken by the person who is executing the test procedures.	Describes the reason for the step.	Describes the expected response of the system to the action specified in the Operator Action column.	Contains additional information such as boundary data, a discussion of the rationale for the step or operator action, or the test strategy underlying the step. May also contain pointers to FDD, CDD, and/or change request documentation.
[Document No.] [Release No.]		[Page No.]		

Fig. 5. Five-column format for specifying test procedures.

TEST INCIDENT REPORT	
TIR #:	DATE:
TEST #:	TESTER:
DESCRIPTION OF PROBLEM:	PROBLEM RECEIVED BY: _____
RECOMMENDED SOLUTION:	
RECOMMENDED BY: _____	
REWORKING NEEDED:	
STEPS REQUIRED TO MAKE OPERATIONAL:	
PRODUCT ASSURANCE MANAGER:	PROGRAMMING MANAGER:

**FILLED  
OUT  
BY  
TESTER**

**FILLED  
OUT  
BY  
SOFTWARE  
ENGINEERING  
DEPARTMENT**

Fig. 6. Test Incident Report (TIR) form.

testing. (It may be useful to attach copies of at least some of the TIR's to the memorandum because, for example, upper level management may wish to get a feeling for the nature of the problems encountered during testing.)

3) Highlights particularly significant problems (in this case, the problems defined in TIR's 37 through 45) and offers a starting point for the programming staff to seek solutions to these problems through a pointer to a specification document (in this case, CDD #1-CS).

Following the TICCB meeting, the Software Engineering Department begins developing solutions to the TIR's on its copy of the development baseline. These solutions are documented on the bottom part of the TIR form shown in Fig. 6. The Software Engineering Department completes its analysis of all of the TIR's prior to the scheduled date of the next TICCB meeting. The Software Engineering Department also

makes coding changes to the development baseline that it believes will correct the problems (and performs tests to see if the problems have indeed been corrected). The Software Engineering Department completes as many of these changes as possible prior to the scheduled date of the TICCB meeting. (If too many changes are still not completed by that date, the Software Engineering Department may request a postponement of the TICCB meeting.) On the scheduled date, another TICCB meeting is convened and the Software Engineering Department formally returns the (updated) development baseline to the Product Assurance Department. The solutions to the TIR's presented at the previous TICCB meeting are discussed. The Software Engineering Department may also present new TIR's generated as a result of its attempt to fix the other TIR's. Following the meeting, the Product Assurance Department writes a memorandum such as that

MEMORANDUM	
TO: Distribution	DATE: 15 May 1995
FROM: Product Assurance Department	CHRON: 95-PAD-105
SUBJ: Release XX Testing Meeting #1	
1.0 Date of Meeting: 14 May 1995	
2.0 Attendees: [Here, the name of each person who attended the meeting and his organizational affiliation are listed.]	
3.0 CCB Action:	
a. Release XX was turned over to the programming staff.	
b. One hundred and five (105) TIR's were turned over and discussed. There was extensive discussion of TIR's 37 through 45 which describe problems related to the new menus defined in CDD #1-C5. The manager of the Software Engineering Department indicated that he would have his staff give particular attention to this set of problems.	
c. It was agreed that the Software Engineering Department would return the Release XX software to the Product Assurance Department on 21 May 1995 for additional testing.	
Distribution: [Here, the name of each person who is to receive a copy of the memorandum is listed. This list generally includes all the meeting attendees and corporate management.]	

Fig. 7. Sample TICCB minutes summarizing the turnover of the development baseline from the Product Assurance Department to the Software Engineering Department.

shown in Fig. 8 that summarizes what happened at the meeting.

The Product Assurance Department then repeats the activities it performed when it first received the development baseline. The cycle of turnovers between the Product Assurance Department and the Software Engineering Department continues until no TIR's remain or until there is corporate agreement that any outstanding TIR's are not sufficiently serious to prevent demonstration of the development baseline to the customer. Each turnover is documented as indicated in Figs. 7 or 8.

As shown in Fig. 4, in-house testing (Step 5) is followed by a demonstration to the customer (Step 6). This demonstration typically consists of execution of our test procedures by the customer at our facility augmented by customer excursions from these procedures. As a result of this demonstration, additional TIR's may be generated that the customer may want fixed before the development baseline is deployed for onsite testing. When the demonstration (which may last several days or longer) has been completed to the satisfaction of the customer, he signs the test procedures. The development baseline is then deployed to the user sites. Testing by our company and/or other customer agents is then performed on the development baseline operating in a live environment. Additional test incidents that the customer may want corrected may result from this testing. When the customer and the user agree that the development baseline is operating satisfactorily in accordance with our and/or the other customer agents' test procedures minus any mutually acceptable discrepancies, the user states in writing that the development baseline has been accepted. At that point, the development baseline becomes the new operational baseline.

61 MEMORANDUM	
TO: Distribution	DATE: 22 May 1995
FROM: Product Assurance Department	CHRON: 95-PAD-129
SUBJ: Release XX Testing Meeting #2	
1.0 Date of Meeting: 21 May 1995	
2.0 Attendees: [Here, the name of each person who attended the meeting and his organizational affiliation are listed.]	
3.0 CCB Action:	
a. Release XX was returned to the Product Assurance Department.	
b. Of the one hundred and five (105) TIR's turned over at the 14 May meeting, one hundred (100) TIR's have been corrected via code changes (TIR's #1 through 87, and 91 through 103).	
c. The manager of the Software Engineering Department stated that TIR #88 was the result of an improper operation of the system by the Product Assurance Department and therefore no corrective action was required. The manager of the Product Assurance Department said that he would correct the pertinent test procedures to provide appropriate clarification to the testers.	
d. The manager of the Software Engineering Department submitted TIR's 106 through 125. He indicated that solutions and associated code had been developed for TIR's 106, 119, and 122 through 125. He also indicated that solutions for the remaining new TIR's have not yet been developed. There was some discussion about TIR 119 which the manager of the Software Engineering Department felt may not really be a problem because the specification, FDD #2-185, was vague in the area of concern. The manager of the Software Engineering Department stated that he wrote TIR 119 to obtain clarification on the matter. It was decided that the problem cited in TIR 119 was indeed a problem.	
e. It was agreed that the Product Assurance Department would return the Release XX software to the Software Engineering Department on 30 May 1995.	
Distribution: [Here, the name of each person who is to receive a copy of the memorandum is listed. This list generally includes all the meeting attendees and corporate management.]	

Fig. 8. Sample TICCB minutes summarizing the turnover of the development baseline from the Software Engineering to the Product Assurance Department.

#### V. KEEPING THE CUSTOMER SATISFIED--THE CONFIGURATION CONTROL BOARD (CCB) AND THE MAINTENANCE PROCESS

Following the establishment of the new operational baseline, the user employs it in his live operational environment until the next major release replaces it. During this operational phase, the user may require that maintenance be performed on the software. One cause of this maintenance requirement may be the discovery of latent defects in the software, i.e., software performance that fails to meet the user's previously stated requirements. Other causes of maintenance action are a change in the user's needs or a desire to enhance his system. Each maintenance requirement is documented in the form of an operational incident report. Operational incident reports are forwarded to the Maintenance Configuration Control Board (MCCB) for processing.

The MCCB consists of CTEC and customer representatives. Jointly chaired by our project management and the

customer's project management, it meets regularly (typically weekly) to process operational incidents submitted by users of the currently deployed operational baseline. In response to these incidents, the MCCB takes one of the following actions:

1) Determines that the incident was the result of improper user operation of the system. In this case, the user is informed of the proper way to use the system, and the incident report is simply archived.

2) Determines that the incident was either 1) the result of proper user operation and therefore represents a system deficiency or 2) an operation not provided by the current system and therefore represents a system enhancement or a changed user need. In both cases, the MCCB approves a change request that, depending on the nature of the change, results in one of the following dispositions:

a) The change is within the scope of the existing maintenance contract. In this case, the change is submitted to the Software Engineering Department for coding and incorporation into the next release of the operational baseline.

b) The change is not within the scope of the existing maintenance contract. In this case, the change is filed for incorporation into a subsequent contract (i.e., customer requirements statement) for eventual incorporation into a future major release.

c) The change is needed to correct a problem that is making the operational baseline inoperative or unable to satisfactorily support operations. In this case, an emergency patch to the operational baseline is authorized by the MCCB. The Software Engineering Department prepares the patch code, which is then submitted to the Product Assurance Department for testing. If this testing indicates that the patch code corrects the problem, the patch code is sent to the affected user sites where it is incorporated into the operational baseline. This patch code is also incorporated into the development baseline.

Each MCCB meeting is documented by a set of minutes similar in nature and form to those of the TICCB (see Figs. 7 and 8). As was the case with the TICCB minutes, the minutes of the MCCB are recorded, archived, and reported by a member of the Product Assurance Department.

#### VI. SUMMARY AND CONCLUSIONS

In the preceding sections, an organization and procedures for making software visible, operational, and maintainable have been described. The organization emphasizes an independent product assurance group. During the software development and change control process, the transition from a statement of customer needs to software code is accomplished through a series of design reviews that provide visibility to the developing software and enhance its maintainability through the establishment of traceability. The testing cycle, which determines that operating software is consistent with customer needs, ensures that the software works. During the operational period, the Maintenance CCB ensures that the software as changed is visible, operational, and retains its maintainability.

The foregoing practices have been successful for our company, and we feel that they can be applied to problems faced by similar small organizations developing and maintaining

software systems. Their adoption/adaptation can enhance the likelihood of successfully meeting the software engineering project management challenge. We feel that our experience is further confirmation of the hypothesis stated in [6] that "large-project software engineering procedures can be cost-effectively tailored to small projects."

#### 62 REFERENCES

- [1] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.
- [2] E. H. Bersoff, V. D. Henderson, and S. G. Siegel, *Software Configuration Management: An Investment in Product Integrity*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
- [3] W. Stallings, "A matrix management approach to system development," in *Proc. 19th Annu. ACM/NBS Tech. Symp.: Pathways to System Integrity*, June 1980, pp. 41-48.
- [4] W. Bryan, S. Siegel, and G. Whiteleather, "Auditing throughout the software life cycle: A primer," *IEEE Computer*, vol. 15, pp. 57-67, Mar. 1982.
- [5] —, "An approach to software configuration control," in *Performance Evaluation Review (1981 ACM Workshop/Symp. Measurement and Evaluation of Software Quality)*, Mar. 25-27, 1981, vol. 10, no. 1, Spring 1981, pp. 33-47.
- [6] B. W. Boehm, "An experiment in small-scale application software engineering," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 482-493, Sept. 1981.



William Bryan received the Ph.D. degree in computer science from George Washington University, Washington, DC, in 1976.

He is currently a senior staff member in the Intelligence Systems Division of CTEC, Inc., McLean, VA. He has been actively involved at CTEC in the application of software configuration management and other product assurance disciplines, including quality assurance, verification and validation, and test and evaluation. This activity has included detailed technical audits of

software and documentation, disciplined control of changes to software and documentation, and acceptance testing of software systems, all for systems ranging in size from small to very large. He has lectured extensively on software product assurance, both nationally and internationally. His lecturing is based on his considerable experience in the actual practicing of software product assurance. He has over 23 years experience in the software engineering profession, and has worked in the specification, development, and maintenance of military command and control systems, industrial process control systems, and large database management systems.



Stanley Siegel received the Ph.D. degree in theoretical nuclear physics from Rutgers University, New Brunswick, NJ, in 1970.

He is currently the Technical Director of the Intelligence Systems Division of CTEC, Inc., McLean, VA. A 100-person company, CTEC develops software systems that support military and intelligence operations. CTEC also provides independent verification and validation (V&V) services which include application of the software life cycle management techniques described in his textbook on software configuration management. He has worked in the computer field in various areas, including systems programming and automated support for national level military command and control systems. During the past seven years, he has been at CTEC performing and teaching software product assurance. He lectures internationally on this subject, which includes configuration management, quality assurance, auditing, and testing.



# Reviews, Walkthroughs, and Inspections

GERALD M. WEINBERG AND DANIEL P. FREEDMAN

**Abstract**—Formal technical reviews supply the quality measurement to the "cost effectiveness" equation in a project management system. There are several unique formal technical review procedures, each applicable to particular types of technical material and to the particular mix of the Review Committee. All formal technical reviews produce reports on the overall quality for project management, and specific technical information for the producers. These reports also serve as an historic account of the systems development process. Historic origins and future trends of formal and informal technical reviews are discussed.

**Index Terms**—Project management, software development management, technical reviews.

## THE PROBLEM OF CONTROLLING TECHNICAL INFORMATION

ANY CONTROL system requires reliable information. A project management system normally obtains its information by two quite different routes, as indicated in Fig. 1. *Cost and schedule information* comes in channels relatively independent of the producing unit, and can thus be relied upon to detect cost overruns and schedule slippages. *Evaluation of technical output*, however, is often another matter.

If project management is not in a position to evaluate technical output directly, it must rely on the producing unit's own evaluation—a dangerous game if that unit is malfunctioning. If the unit is technically weak in a certain area, the unit's judgment will be weak in the same area. Just where the work is poorest, the evaluation sent to management will be least likely to show the weakness.

But even if the producing unit is not technically weak, the problem of unreliable information persists because of information-overload. As a unit overloads, inadequate supervision may affect work quality—while at the same time affecting the quality of the evaluation. The unit *wants* to be done on schedule and *wants* the work to be correct. Under pressure, any human being will see what is wanted instead of what exists. Just when it is needed most, this control system utterly fails.

## THE ROLE OF THE FORMAL TECHNICAL REVIEW

Formal technical reviews come in many variations, under many names, but all play the same role in project management, as indicated in Fig. 2. As in Fig. 1, the producing unit controls its own development work, perhaps even conducting informal reviews internally. At the level of the producing unit, in fact, the use of the formal technical review requires no

63

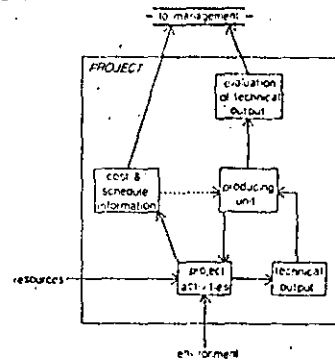


Fig. 1. Management's view of the output of a programming effort.

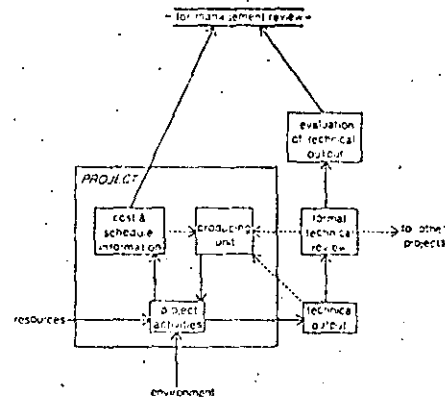


Fig. 2. The place of the formal technical review.

change, which simplifies its introduction as a project management tool.

As the diagram shows, the formal review is conducted by people who are *not part of that producing unit*. Hopefully, these are people who have no conscious or unconscious reason for favoring or disfavoring the project's work. Moreover, their report—the technical review summary report—goes to management, thus providing *reliable information* to be used in *management reviews* of the project.

## MANAGEMENT REVIEWS VERSUS TECHNICAL REVIEWS

Fig. 2 also illustrates the difference between a *technical review* and a *management review*, sometimes called a "project

Manuscript received January 5, 1983.  
G. M. Weinberg is with Weinberg and Weinberg, Rural Route Two, Lincoln, NE 68505.  
D. P. Freedman is with Ethnotech, Inc., P. O. Box 6627, Lincoln, NE 68506.

review" or some similar name. The technical review committee is staffed by technical people and studies only technical issues. Its job is to put the evaluation of technical output on the same reliable basis as, say, cost and schedule information to management. Using both sorts of information, management can now make informed judgments of what is to be done in controlling the project.

It should also be noted that most "project control" systems do not concern themselves with the accurate and reliable evaluation of the quality of technical output. Instead, they concern themselves with measuring what can be measured *without* technical review, assuming, more or less, that one module of 300 lines of code is just like any other. If that assumption of quality is correct, then these systems can provide excellent management information for project control.

If that assumption is not correct, however, then only the "cost" side of the "cost effectiveness" ledger has any meaning. Under such conditions, even the best project control systems can provide only an illusion of control. The consequences are familiar enough—missed schedules, cost overruns, unmet specifications, inadequate performance, error-prone production, and huge and never ending maintenance.

#### REVIEW REPORTS AND PROJECT MANAGEMENT

Whatever goes on *inside* it, the major project control function of *any* review is to provide *management* a reliable answer to the fundamental question:

*Does this product do the job it is supposed to do?*

Once any piece of work has been reviewed and accepted, it becomes part of the system. Subject to a very small risk factor, it is

- 1) complete,
- 2) correct,
- 3) dependable as a base for related work, and
- 4) measurable for purposes of tracking progress.

Without reviews, there are no *reliable* methods for measuring the progress of a project. *Sometimes* we get dependable reports from the producers themselves, but *sometimes* is not good enough. No matter how good their *intentions*, producers are simply not in a position to give consistently reliable reports on their own products.

For small, simple objects, with well-intentioned, competent producers, there is some chance of success without reliable progress measures. As projects grow larger and more complex, however, the chance of some self-report being overly optimistic becomes a certainty.

Whatever the system of formal reviews, the review reporting serves as a formal commitment by technically competent and unbiased people that a piece of work is complete, correct, and dependable. The review report states as accurately as possible the completeness and acceptability of a piece of software work, be it specifications, design, code, documentation, test plan, or whatever.

By themselves, these review reports do not guarantee that a project will not end up in crisis or failure. It is up to the management of the project to use the information in the review reports to make management decisions needed to keep the project on track. Well done review reports are not suf-

ficient to make a project succeed, though poorly done reviews, or no reviews at all, are sure to get a project in trouble—no matter how skilled the management or how sophisticated the project management system.

## 64 TYPES OF TECHNICAL REVIEW REPORTS

The one report that is always generated by a *formal* review is the *technical review summary report*. This report carries the conclusions of the review to management, and thus is the fundamental link between the review process and the project management system.

Other reports *may* be generated. Issues raised that must be brought to the attention of the producers are placed on a *technical review issues list*.

If issues are raised about something other than the reviewed work itself, a *technical review related issues report* is created for each issue.

On occasion, an organization will institute some *research report*, such as a detailed breakdown of standards used and broken, or a report of hits and misses on a checklist.

Those cases where the review leader has to give a report of a failed review (not a failed product) lead to a *review process report*, the form and content of which will be unique to the situation and organization. For instance, on delicate matters the review process report may be verbal.

Other participants may also report on the process of the review itself. For instance, one or more participants might want to object to the behavior of the review leader.

#### THE REVIEW SUMMARY REPORT

For effective project management, review summary reports must identify three items:

- 1) What was reviewed?
- 2) Who did the reviewing?
- 3) What was their conclusion?

Fig. 3 shows a widely used format containing these items. Although formats vary, the summary should generally be confined to a single page, lest its conclusion be lost in a forest of words.

#### THE TECHNICAL REVIEW ISSUES LIST

Whereas the summary report is primarily a report to management, the issues list is primarily a report to the producers. The issues list tells the producers *why* their work was not fully acceptable as is, hopefully in sufficient detail to enable them to remedy the situation.

The issues list is primarily a communication from one technical group to another. It is not intended for nontechnical readers and therefore need not be "translated" for their eyes. Moreover, it is a *transient* communication, in that once the issues are resolved, the list might as well disappear. (We exclude, for the moment, research use of the list). Therefore, the issues list need not be fancy, as long as it is clear.

Practices vary, but among our clients, management does not routinely get the issues list. The summary report already contains, in its assessment of the work, a weighted opinion of the seriousness of the issues, so management need not be burdened with extra paper and technical details.

TECHNICAL REVIEW SUMMARY REPORT

REVIEW NUMBER 1.2.2.1 STARTING TIME 10:00  
 DATE MAY 3, 1974 ENDING TIME 10:10

WORK UNIT IDENTIFICATION EDIT-FILE-TXT TRANSACTION EDITOR  
 PRODUCED BY PAOLILLO, NOWACKI, AND GARFIELD  
 BRIEF DESCRIPTION SOITS AND FORMAT INPUT TRANSACTIONS FOR  
UPDATING EX-PARTS CATALOG

MATERIALS USED IN THE REVIEW: (check here if supplementary list)  
 IDENTIFICATION DESCRIPTION  
 COMPILED CODE LISTING  ALL SYNTAX ERRORS CORRECTED  
 TEST FILE LISTING  INCLUDING ERROR CASES  
 TEST RESULTS LISTING  SIMULATED BY DEVELOPER, NOT USER  
 CRITICAL ASSUMPTION LIST  ANY OF THESE WOULD REQUIRE MAJOR  
 REVISION OF THE SYSTEM IF CHANGED,  
 IN THE OPINION OF THE DEVELOPER.

PARTICIPANTS:

NAME	SIGNATURE	I.D.
LEADER <u>J. YAO</u>	<i>J. Yao</i>	-----
RECORDED <u>A. MARZETTA</u>	<i>A. Marzetta</i>	-----
3. <u>P. SCHWARTZ</u>	<i>P. Schwartz</i>	-----
4. <u>L. BARNAN</u>	<i>L. Barnan</i>	-----
5. <u>L. NOWACKI</u>	<i>L. Nowacki</i>	-----
6. <u>P. THOMPSON</u>	<i>P. Thompson</i>	-----
7. _____	_____	-----

APPRAISAL OF THE WORK UNIT:  
 ACCEPTED (no further review)  
 as to  
 with minor revisions  
 NOT ACCEPTED (further review required)  
 major revisions  
 rewrite  
 review not completed

SUPPLEMENTARY MATERIALS PRODUCED:  
 IDENTIFICATION AND/OR IDENTIFICATION  
 ISSUES LIST 1.2.2.1-1000-1000-1000-1000  
 RELATED ISSUES LISTS field edit  
 OTHER \_\_\_\_\_

Fig. 3. A technical review summary report.

The issues list need not be *concealed* from management, but when managers routinely receive lists of issues, they naturally try to use the information. For example, they may count issues as a means of evaluating producers or reviewers—a practice which tends to undermine the quality of future reviews.

Another common subversion of the review process is the attempt to make the issues list into a *solutions list*. The job of the review committee is to raise issues; the job of the producing unit is to resolve them. A review committee is generally no better at resolving issues than a producing unit is at raising them. Management may want to see issues lists from time to time to ensure that they are remaining issues lists, rather than solutions lists.

#### THE TECHNICAL REVIEW RELATED ISSUES REPORT

A related issue is something that comes up in the course of a review that does not happen to be the principal reason for the review. Examples of related issues might be:

- 1) a typographical error in a related document;
- 2) a hidden assumption in the specifications that makes part of one module obsolete;
- 3) a flaw in the original problem statement that makes the entire project plan invalid.

If an organization cannot handle issue 1) without alerting the management chain, it is probably in as bad a shape as an organization that handles issue 3) *without* alerting the manage-

ment chain. The principal project management problem is with middle issues, such as 2). Such issues have always been troublesome to project control systems. When they are detected, the related issue report is a way of notifying *someone* who ought to be in a position to do something about them.

Because a related issue is, by definition, a deviation from smooth product development, there is really no way to develop a standard practice for handling all situations. A related issue report often descends like a bolt from the blue on some people who may not even know that a review is taking place. If it is not communicated in some standard, official form, people may not even recognize it. Therefore, if we want to keep related issue reports from passing directly into the wastebasket, we have got to give them *some* official status.

The mildest approach is to use a standard *transmittal sheet*, identifying the source of the material and attached to the actual communication, which may take any convenient form. Some organizations prefer a formal follow-up system that requires that each related issue report receive a reply within a few days. Another approach is to send the related issue report through the appropriate manager, leaving any action or follow-up decision on the managerial level.

#### HISTORICAL ANALYSES

Some of the information obtained from an historical analysis of review reports can be extremely specific. For instance, many organizations classify the types of problems turned up in each review and tabulate the frequency of each type. A similar tabulation is made of errors that slip through the review only to be caught at a later stage.

Comparison of these tabulations—in total, by review group, and by producer—provides clear guidance for future educational and reviewing practices. It is essential, however, that this information be used for improvement of project management, not for punishment of individuals, lest the whole scheme backfire and produce better methods of concealing errors and deficiencies.

To illustrate appropriate use of such historical analyses, let us say that most of the flaws detected during code reviews centered around the module interfaces. If this deficiency was project-wide, the training department could set up special training for everyone, guided by the specific types of interfacing errors recorded in the review reports.

Or perhaps the interfacing errors, upon analysis, reveal a weakness in project standards concerning interfaces. Whatever the problem, the historical records should first make it visible, then make it measurable, and finally help narrow it down to its true source.

#### REVIEWS AND PHASES

Any time after a project begins, an accurate, complete review report history can be compared with the schedule projected at the beginning of the development cycle. In which phases did the estimated time match the actual time? Where did the deviations occur? Were the deviations caused by problems in development? Were they mistakes in the original estimate?

Such historical information is obviously essential if project

management is to improve from project to project. Yet such information will be meaningless if the "phases" of the project plan do not correspond to units of work marked at both beginning and end by reviews.

In order for any project control system to work, the system life cycle must be expressed in terms of measurable phases—some meaningful, reviewable product that represents the end of one phase and the beginning of the next. If there is nothing that can be reviewed, then nothing has been produced, and if nothing has been produced, how can it be controlled?

#### VARIETIES OF REVIEWED MATERIALS

Much of the earliest public discussion of reviews focused on the varieties of *code reviews*, rather than reviews of other materials produced in the life cycle. In the early history of software development, we were primarily concerned with code accuracy, because the coding seemed to be the major stumbling block to reliable product development. As our coding improved, however, we began to see other problems that had been obscured by the tangle of coding errors.

At first we noticed that many of the difficulties were not coding errors but design errors, so more attention was devoted to *design reviews*. As these techniques begin to be effective at clearing up design problems, the whole cycle starts again, for we notice that design is no longer the major hurdle.

In many of these cases, we never clearly understood the problem the design was attempting to solve. We were solving a *situation*, not a problem. Currently, increased emphasis is being placed on the analysis process, which becomes the next area of application of technical reviews—*specification reviews*.

Other types of reviewed material include *documentation*, *test data* and *test plans*, *tools* and *packages*, *training materials*, *procedures* and *standards*, as well as any other "deliverable" used in a system.

Reviews of these materials are conducted not only during development, but also during operation and maintenance of the system.

#### PRINCIPAL VARIETIES OF REVIEW DISCIPLINES

It is possible to conduct a review without any particular discipline decided in advance, simply adjusting the course of the meeting to the demands of the product under review. Many reviews are conducted in just this way, but over time special disciplines tend to evolve which emphasize certain aspects of reviewing at the expense of others.

For instance, many of the best known review disciplines are attempts to "cover" a greater quantity of material in the review. The "inspection" approach tries to gain efficiency by focusing on a much narrower, much more sharply defined, set of questions. In some cases, an inspection consists of running through a checklist of faults, one after the other, over the entire product. Obviously, one danger of such an approach is from faults that do not appear on the checklist, so effective inspection systems generally evolve methods for augmenting checklists as experience grows.

Another way to try to cover more material is by having the

product "walked through" by someone who is very familiar with it—even specially prepared with a more or less formal presentation. Walking through the product, a lot of detail can be skipped—which is good if you are just trying to verify an overall approach or bad if your object is to find errors of detail.

In some cases, the walkthrough is very close to a lecture about the product—which suggests another reason for varying the formal review approach. In some cases, rapid education of large numbers of people may suggest some variation of the formal review.

In a walkthrough, then, the process is driven by the *product being reviewed*. In an inspection, the *list of points to be inspected* determines the sequence. In a plain review, the order is determined by the *flow of the meeting as it unfolds*. In contrast to these types, the various kinds of "round-robin" reviews emphasize a *cycling through the various participants*, with each person taking an equal and similar share of the entire task.

Round-robin reviews are especially useful in situations where the participants are at the same level of knowledge, a level that may not be too high. It ensures that nobody will shrink from participation through lack of confidence, while at the same time guaranteeing a more detailed look at the product, part by part.

#### REAL VERSUS IDEAL REVIEWS

Although many "pure" review systems have been described, people who observe actual reviews will never find one following all the "rules." By examining some of the real advantages and disadvantages of one of these "pure" systems, we can understand why every real review system involves aspects of all the major varieties. We will use the walkthrough as our example, but any system could be used to illustrate the same points.

With a walkthrough, because of the prior preparation of the presenter, a large amount of material can be moved through rather speedily. Moreover, since the reviews are far more passive than participating, larger numbers of people can become familiar with the walked through material. This larger audience can serve educational purposes, but it also can bring a great number of diverse viewpoints to bear on the presented material. If all in the audience are alert, and if they represent a broad cross section of skills and viewpoints, the walkthrough can give strong statistical assurance that no major oversight lies concealed in the material.

Another advantage of the walkthrough is that it does not make many demands on the participants for preparation in advance. Where there are large numbers of participants, or where the participants come from diverse organizations not under the same operational control, it may prove impossible to get everyone prepared for the review. In such cases, the walkthrough may be the only reasonable way to ensure that all those present have actually looked at the material.

The problems of the walkthrough spring rather directly from its unique advantages. Advance preparation is not required, so each participant may have a different depth of

understanding. Those close to the work may be bored and not pay attention. Those who are seeing the work for the first time may not be able to keep up with the pace of presentation. In either case, the ability to raise penetrating issues is lost.

#### WHY THERE IS SO MUCH VARIETY IN REVIEWS

Although all reviews occupy the same role in project management as a control system, managers are justifiably confused by the great variety found in technical review practices. The practice of technical review differs from place to place for a variety of reasons, the principal ones being:

- 1) different external requirements, such as government contract provisions;
- 2) different internal organizations, such as the use or nonuse of teams;
- 3) continuity with past practices.

Continuity is probably the strongest reason. When it comes to social behavior, people tend to resist changing what they already do, even if it does not seem exceptionally productive in today's environment. In many project management systems, formal technical reviews have been introduced as a new form of some old practice, perhaps because it was easier to introduce reviews in this way.

#### HOW REVIEWS EVOLVED

The idea of reviews of software is as old as software itself. Every early software developer quickly came to understand that writing completely accurate programs was too great a problem for the unaided human mind—even the mind of a genius. Babbage showed his programs to Ada Lovelace, or to anyone else who would review them. John von Neumann regularly submitted his programs to his colleagues for review.

These reviews, in our terms, were *informal* reviews, because they did not involve formal procedures for connecting the review reports to a project management system. Informal review procedures were passed on from person to person in the general culture of computing for many years before they were acknowledged in print. The need for reviewing was so obvious to the best programmers that they rarely mentioned it in print, while the worst programmers believed they were so good that *their* work did not need reviewing.

Around the end of the 1950's, the creation of some large software projects began to make the need for some form of technical reviewing obvious to management all over the world. Most large projects had some sort of reviewing procedures, which evolved through the 1960's into more formalized ideas.

In the 1970's, publication espousing various review forms began to appear in the literature. For those interested in a history of publication, a bibliography appears in Freedman

and Weinberg [1]. Publications, however, tend to conceal the grass-roots origin of reviews, giving the impression that they were "invented" by some person or company at a certain time and place.

#### WHERE REVIEWS ARE GOING

Today, the evolution of reviewing procedures continues, primarily on an experiential basis within projects. Reviews are a partial formalization of a natural social process, arising from the superhuman need for extreme precision in software. Therefore, the "science" of reviewing is a *social* science, and it is difficult to make general, quantifiable statements that apply to all reviews.

Some experimental work has been done on reviews, but these experiments generally suffer from the following problems:

- 1) Only one or two narrowly defined review procedures are examined.
- 2) Reviewers are novices in the procedures used.
- 3) The environment is significantly different from that of a real software development or maintenance environment.

Field reports overcome items 2) and 3), but introduce the problem of experimental control. Nevertheless, many of these reports indicate that effective project management is not possible without the technical review, in one form or another. These reports are sometimes puzzling to managers in other organizations, who have "tried reviews," but who have failed to overcome some of the human problems of changing entrenched social practices.

The best evidence for the effectiveness of reviews is that their use continues to spread. A body of practical knowledge has grown with this spread, particularly concerning the problems associated with starting a system of reviews. We anticipate that most future development of review technology will arise from such on-the-job experiments, rather than any theoretical or laboratory work.

#### REFERENCES

- [1] D. P. Freedman and G. M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*, 3rd ed. Boston, MA: Little, Brown and Company, 1982. (Because this reference contains an extensive bibliography, we are omitting further references here.)

Gerald M. Weinberg, photograph and biography not available at the time of publication.

Daniel P. Freedman, photograph and biography not available at the time of publication.

# Software Engineering Project Standards

MARTHA BRANSTAD AND PATRICIA B. POWELL

68

**Abstract**—Software Engineering Project Standards (SEPS) and their importance are presented in this paper by looking at standards in general, then progressively narrowing the view to software standards, to software engineering standards, and finally to SEPS. After defining SEPS, issues associated with the selection, support, and use of SEPS are examined and trends are discussed. A brief overview of existing software engineering standards is presented as the Appendix.

**Index Terms**—Project management, software development, software engineering, software engineering standards, software management, software standards.

## I. INTRODUCTION

A STANDARD can be: 1) an object or measure of comparison that defines, represents, or records the magnitude of a unit, 2) a canonical form or characterization that establishes allowable tolerances or constraints for categories of items, 3) a degree or level of required excellence, confidence, assurance, or attainment. Fig. 1 displays this breakdown and examples from each category. Measurement standards provides a metric by which an entity can be measured or compared, e.g., a standard meter or standard time. An interchange standard provides a norm for product compatibility, e.g., standard light socket or language standard. A performance standard provides a categorization or descriptive framework, e.g., a grading system.

Standards are definitional by nature, either established to further understanding and interaction, or as observed norms of exhibited characteristics or behavior. Standards are formulated when there is motivation for control of variability. Many institutionalized standards have been established to enable commercial interaction and division of labor (e.g., standards for weights, machine components, etc.). Other standards clarify quality aspects implied by the use of given terms (e.g., U.S. meat grades).

Early software standardization efforts emphasized interchange of products. Standards were defined to control variability and to facilitate software transportability. Programs and data produced or accumulated at one site needed to be moved to computers, perhaps produced by a different manufacturer, at another site. The Code for Information Interchange (ASCII) standard (Federal Information Processing Standard 1-1) and the Cobol language standard (Federal Information Processing Standard 21-1) are examples of standards established to facilitate interchange. In contrast, current software engineering standards are motivated primarily by a desire to establish levels of confidence or definitional aspects of software rather than to foster global interchange of software [27]. Many of the current software standardization efforts are motivated by

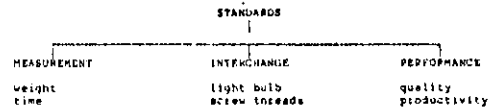


Fig. 1. Categories of Standards.

management's desire to control software quality and improve productivity in order to achieve better control of economic issues, for example, resource estimation (how to estimate software development cost, development time, required resources, and reliability).

Efforts to establish software engineering standards for large, heterogeneous populations must reach a compromise between the desire for specific, detailed standards and the diverse needs of the constituent bodies. Software engineering standards issued by IEEE, Federal Information Processing Standards (FIPS), and the American National Standards Institute (ANSI) are examples. Particularly when quality control rather than product interchange is the underlying motivation, "global" standards tend toward the definition of general frameworks, planning guides, or tailorable standards. These are carefully defined to serve as a basis for communication, general quality control, and the establishment of norms of good practice, while providing leeway for the use of diverse development techniques and approaches.

The rather general, global standards serve a very real need for industry-wide norms and definitions; however, for use within a software development project more specific standards are appropriate. It is in this domain, that of a specific software development project, that Software Engineering Project Standards (SEPS) should be established and used. Within the project, the need for specificity does exist, for materials must be communicated and shared among members of the development team. SEPS serve the needs for both quality control and information interchange to support division of labor. SEPS can be very specific since the standards are being established for a particular project with known characteristics. SEPS are (or should be) the embodiment of project development policy. They should establish the development methods to be used; the specific requirements, design, and coding techniques and languages; the verification, validation and testing (VV&T) approach; the form and type of records to be kept and controlled; and the official documents that should be produced. The specific nature of the SEPS and the limited domain in which they are applied create both opportunity and challenges which are discussed later in the article.

To synopsise the preceding discussion, Software Engineering Project Standards are norms of required practice established

Manuscript received July 13, 1982.  
The authors are with the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC 20234.

by the project manager in order to produce a software system of uniformly high quality and to facilitate communication and interchange within the project. The SEPS comprise a collection of standards which cover both management and technical aspects of software development and provide the general framework in which software development is performed. When defined in their entirety, SEPS represent an embodiment of project development policy.

## II. ISSUES

69

The successful establishment and use of SEPS involves significant understanding and insight into the state of current software technology, human nature, people's ability to deal with change, and the goals of the particular organization and project. A number of specific issues associated with the selection, introduction, support, and use of SEPS are presented below. See [10], [26], [29] for additional discussion of standards issues.

### *Standards and Measurability*

By their very nature, standards involve measurement. Since a standard is a model or rule against which other objects are compared or measured, it is essential that it be possible to determine if the candidate complies with the standard or is within an acceptable tolerance of the standard. Although this may seem obvious, it is often overlooked. This is particularly true in the software engineering arena where the desire for increased quality and productivity is a strong force behind SEPS. Software quality is usually defined as a collection of characteristics, e.g., efficiency, maintainability, testability [18], with the importance of each specific characteristic varying from project to project. However, the quality characteristics are difficult to measure directly. Two different approaches to this dilemma have arisen. The first approach standardizes those properties which are amenable to measurement even though it is recognized that the properties are secondary or support characteristics and do not guarantee quality software. Module size, control structure complexity, and naming conventions are examples. The second approach concentrates upon the process by which the software product is produced rather than on the characteristics of the product itself. To effect the second approach, specific steps in the development process are standardized both with respect to their occurrence and to the techniques used to accomplish the step. Although both types of software engineering standards are used in the collection comprising the SEPS, the process standards are currently viewed as the most important [3], [30].

A mild warning should be given concerning measurement of project characteristics. People are very effective at almost unconscious behavior modification in response to what is perceived as desirable or rewarded behavior. If lines of code are measured and coders believe that more is better, more will be produced. Only important and meaningful characteristics should be emphasized.

### *Standards and Technology*

Software engineering standards are dependent upon the technology that they represent and serve. They cannot outpace the technology, neither should they curtail or suppress it.

The standards should incorporate stable technology which is available for distribution and use. Technical feasibility alone is insufficient; the underlying techniques and the support required to implement the standards must be readily available. Standards for dynamic technology involve a fundamental conflict between the stability implied by the standard and the change inherent in the technology. Software engineering standards must deal with this issue. Not only is computer hardware still evolving, but the software engineering tenets that the standards should embody are still under development. Although there is general agreement about the more fundamental principles, there are many open issues. The solutions have a significant likelihood of incorporating new insights, rather than being only further refinements of existing "truths." The need for future modification must be anticipated when establishing software standards. Since the implied duration is more limited for project standards, this problem of adapting the standards to a dynamic technology is somewhat less severe than for global standards. Each project manager has the responsibility for defining SEPS; however, the cost associated with introducing SEPS that differ significantly from those used previously must be addressed. For both software engineering standards and SEPS the cost of change must be weighed against the cost of not changing.

### *Introduction and Implementation of Standards*

Management reaps the potential benefits of SEPS: decreased variability, increased product quality, increased worker productivity, facilitated communication, and better control [2]. Management must make a definite commitment for a SEPS effort to come to fruition and be effective. The introduction of SEPS will, in most cases, involve a change in the manner in which work has been performed previously [31]. Change is difficult for most people and hence they tend to resist it. Therefore, anytime a change is made and it is desired that it be successfully incorporated, effort must be expended to facilitate the transition. The change, in this case the introduction of SEPS, must be motivated. (It is assumed that the SEPS were carefully selected so that they are reasonable and understandable for the given project environment.) An education program should be initiated. People must understand the SEPS in order to use them. If the SEPS represent a significant departure from previous software development procedures, they should be introduced in a phased manner. Automated support for the SEPS should be provided. Software tools that actually embody the standards are most effective. Structure preprocessors and compilers are examples. When appropriate automation is available, it becomes easier to perform the work the standard way than by any alternative means. In such instances, standards audit or enforcement becomes transparent, since the development process incorporates the standard.

### *Global Software Engineering Standards Versus Local SEPS*

In most cases, the project manager does not have complete license in defining the SEPS since at least a portion of the standards are imposed upon him by the organization in which the project exists and the client for whom the work is being done. In addition, the industry is developing voluntary standards which should be considered. Reconciliation of sometimes

conflicting sets of software standards is an initial step in defining the SEPS to be used during a software development project. Frequently, the process involves tailoring general software engineering standards to provide specific standards for the given project. The limited domain and specific nature of SEPS provides opportunity for the selection of particular development techniques. While agreement upon a single technique is unlikely (and inappropriate) in the global arena, there is economic pressure to have SEPS extend beyond the boundary of a single project (unless the project is extremely large and of significant duration.) The overhead costs for training and automated support necessary to introduce and implement the SEPS successfully are more attractive when amortized over several projects within an organization. To the extent permitted by the compatible nature of projects within an organization, the economics suggest that SEPS should be supported throughout the organization.

#### *SEPS and Software Quality Assurance*

Software quality assurance (SQA) is usually approached by control of the development process. The process is specified by developing standards, while quality assurance auditing determines compliance with the standards. Software standards and SQA are companion processes. SEPS provide the basis from which to perform SQA. When developing SEPS, it is of prime importance to select standards which are quantifiable and hence measurable for an objective audit by the SQA group. The SQA group should be organizationally independent from the development effort to maintain its functional perspective [16].

### III. CANDIDATES FOR SEPS SELECTION

The SEPS should be the embodiment of the development policy for the project. As such, the specific standards established will vary from project to project, but should address key aspects of software development. Since the SEPS are comprised of a collection of standards, each relating to a phase or specific aspect of the development process, the individual standards must be compatible and consistent with one another. For example, coding techniques must be compatible with the language being used. The following sections discuss categories of candidate standards for inclusion in SEPS. A number of software life cycles with somewhat different phases have been defined and used within the industry. For illustrative purposes, a four-phase life cycle is used in the following discussion: requirements, design, construction, and maintenance. Activities which span the entire life cycle are termed overview activities.

#### *Overview Activities*

Three areas of overview activities are candidates for standardization: verification, validation, and testing (VV&T); configuration management; and documentation. VV&T activities take place throughout the life cycle to ensure the correctness and quality of the software. VV&T items that should be defined in the SEPS include: VV&T planning documents, review points, specific verification techniques for each life cycle phase, verification techniques to ensure consistency of products between phases, testing standards including coverage, and records for completed VV&T activities. Configuration management

controls documents and products produced during development and the changes made to them. Aspects of configuration management that should be defined in the SEPS include: version identification codes, change control methods, identification of the documents that are to be controlled, and auditing procedures. Documentation is a key to the success of software development. Two categories of documents exist, development records and user-oriented manuals. Development documentation provides the information needed to manage and control the project. SEPS should define the types of documents required, the material they should include and the formality of the presentation. Management and planning documents are included as a subset of the project documentation standards. The overview activities are particularly critical areas for SEPS since they impact all phases of development and are central to record keeping, communication within the project, management control, and general quality control issues.

#### *Requirements*

The requirements phase begins with a project proposal and ends with a requirements specification document which provides the baseline for product validation. Although the complete requirement development process is not amenable to standardization at this time, at the project level, a specific method for recording the requirement specification can be standardized. The method should support both documentation and analysis of the requirements and should facilitate the development of a complete, consistent, and comprehensible requirement specification. See [1] for a discussion of candidate methods and techniques to be used throughout the development cycle.

#### *Design*

During the design phase, the specified requirements are transformed into detailed designs from which code can be produced. Frequently, this phase is organized as a two-step process with preliminary, high-level designs being produced and verified before being further developed into detailed designs. The SEPS should designate a particular method to be used for design, and should further clarify terminology, structuring, complexity, size, and interface constraints as appropriate to the method selected.

#### *Construction*

The construction phase includes coding, integration, test, and installation of the accepted system. This phase has highly visible products which can be measured against predetermined criteria, e.g., module size. SEPS for this phase should establish the high-level language to be used (including special exceptions concerning the use or exclusion of specific language features), naming conventions, module size, code complexity, desired code structuring, commenting and in-line documentation conventions, and interface constraints. Testing standards for the construction phase should be established as a subset of the VV&T standards. Most organizations have standards which delineate the method for code development. Use of automation to determine compliance (code auditing) is becoming more common. Although much standardization activity centers upon this phase, the characteristics being controlled are



of somewhat less importance to the quality of the final software product than are the proper development of requirements and design.

71

### Maintenance

The maintenance phase starts when the software has been installed and accepted and ends when the software is replaced or discarded. Although different management is usually responsible for the software during the maintenance phase, the SEPS are still important. Studies show [17] that 50 percent or more of maintenance activities are software enhancements, work that could easily be described as development in a "constrained" environment. The remaining maintenance activities can be categorized as corrective (error fixing) and adaptive (e.g., modifications required by a changing hardware for software systems environment.) Throughout all these maintenance operations, the original SEPS should be used and enforced. In many cases, the most significant economic return on the investment in software standards comes during the maintenance phase. Unless the use of SEPS continues, however, the improved control and manageability will degrade with each modification to the software.

### V. TRENDS

There has been a significant increase in interest in software engineering standards in recent years, as indicated by the number of IEEE standards committees and working groups that have been organized. The emphasis on software engineering standards is expected to continue in the future with a collection of global software engineering standards being established by IEEE. For the Federal government, a number of FIPS guidelines for various aspects of software engineering are either recently completed [8], [9], in process, or planned. Updated guidance for software documentation, additional guidance for VV&T and acceptance testing, maintenance guidelines, and recommendations for the use of tools throughout the development process are planned. U.S. military standards continue to exhibit an increased emphasis and concern for software engineering standards. The effort that has gone into the Joint Services Defense Systems Software Development Standard is representative.

It is expected that the future will bring an increased effort in planning, organizing, and standardizing the software development process. Software developers will continue to realize the economic benefits of standardization of their processes while national standards will provide modifiable guidelines that can be adapted to specific projects. The trend toward standardizing parts of the software development process will continue, motivated in part by the use of automation to support the development process.

#### APPENDIX SOFTWARE ENGINEERING STANDARDS

Several national standards producing bodies have established a number of global software standards. Software engineering, however, is a relatively recent area for standardization activity. Nevertheless, some software engineering standards already exist, some have been adapted from hardware standards, and

others are currently in development. Summarized below are global software engineering standards, in use or being drafted, from three national sources of standards: IEEE standards, FIPS, and military standards. These software engineering standards are candidates for use during software development and hence impact the SEPS to be established by the project manager. The summaries are extracted from materials available in the Fall of 1983.

#### IEEE Software Engineering Standards

##### IEEE Standard for Software Quality Assurance Plans (ANSI/IEEE Std. 730-1981).

"The purpose of the standard is to provide uniform, minimum acceptance requirements for preparation and content of Software Quality Assurance Plans. This standard applies to the development and maintenance of critical software. . . . For noncritical software, or for software already developed, a subset of the requirements of this standard may be applied" [14]. The standard includes the following major topics: purpose; referenced documents; management; documentation; standards, practices, and conventions; review and audits; configuration management; problem reporting and corrective action; tools, techniques, and methodologies; code control; media control; supplier control; and records collection, maintenance, and retention.

##### IEEE Standard Glossary of Software Engineering Terminology (ANSI/IEEE Std. 729-1983).

The document is a glossary of terms in general use in the field of software engineering [13].

##### IEEE Guide for Software Requirement Specifications (IEEE Std. 830-1983).

"This document is a guide for writing software requirements specifications. It describes the necessary content and qualities of a good Software Requirements Specification (SRS) and presents a prototype SRS outline" [15]. The guideline discusses what a requirement is; qualities of requirements such as unambiguity; completeness, verifiable, consistency, and traceability; a typical outline for a SRS; the evolution of a SRS; methods used to express software requirements; and tools for developing a SRS.

##### IEEE Standard for Software Configuration Management Plans (IEEE Std. 828-1983).

"The purpose of this standard is to provide minimum requirements for the preparation and content of Software Configuration Management (SCM) Plans. . . . This standard applies to the development and maintenance of any kind of software" [11]. The standard discusses the SCM environment overview, i.e., system description, life cycle, and SCM roles, responsibilities, and interfaces; SCM organization, e.g., organizational structures responsibilities and authorities; SCM activities; tools; releases and libraries; SCM throughout the life cycle; and SCM resource requirements.

##### IEEE Standard for Software Test Documentation (ANSI/IEEE Std. 829-1983).

"This standard describes a basic set of test documents

which are associated with the process of analyzing computer programs in order to detect faults and estimate the risk of failure. . . . It is applicable to commercial, scientific, or military, software which runs on any digital computer." [12] The basic documents discussed are the test plan, test design specification, test case specification, test procedure specification, test item transmittal form, test log, test incident report, and test summary report.

New IEEE standards working groups are continually forming. Those which have project authorization are: *A Standard for Software Reliability Measurement* (P982), *A Guide for Software Quality Assurance* (P983), *A Guide for the Use of Ada as a PDL* (P990), *Software Engineering Standards Taxonomy* (P1002), *A Standard for Software Unit Testing* (P1008), *A Standard for Software Plans* (P1012), *A Guide for Software Design Documentation* (P1016), and a revision to *A Standard for Software Quality Assurance Plans* (P730-1).

#### *Federal Information Processing Standards and Guidelines*

*Guidelines for Documentation of Computer Programs and Automated Data Systems* (FIPS PUB 38).

This guideline addresses the content of ten documents for the development phase of the life cycle. The documents are: functional requirements, data requirements, system/sub-system specification, program specification, database specification, user's manual, operations manual, program maintenance manual, test plan, and test analysis report [6].

*Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase* (FIPS PUB 64).

This guideline provides a basis for determining the content and extent of documentation for the initiation phase of software development. Guidance is given for the following document types: Project Request, Feasibility Study, and Cost/Benefit Analysis [7].

*Guideline: A Framework for the Evaluation and Comparison of Software Development Tools* (FIPS PUB 99).

This document is designed to be used as a reference and suggests a framework which aids in identifying, discussing, evaluating, and comparing software development tools. It is recommended for use when acquiring or implementing tools, developing policies or procedures for tools, and reviewing the current use of tools [8].

*Guidelines for Lifecycle Validation, Verification, and Testing of Computer Software* (FIPS PUB 101).

"This guideline recommends that validation, verification, and testing (VV&T) be performed throughout the software development lifecycle. The selection and use of a collection of validation, verification, and testing techniques to meet project requirements is presented. The guideline explains how to develop a VV&T plan which will fulfill a specific project's VV&T requirements" [9].

#### *U.S. Military Standards*

*Specification Practices* (MIL-STD-490).

The standard specifies how a specific type of computer program development specification should be prepared [24].

*Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs* (MIL-STD-483).

The standard provides for a configuration management plan, configuration identification, baselines, change control, and audits; it is hardware oriented [23].

*Technical Reviews and Audits for Systems, Equipment, and Computer Programs* (MIL-STD-1521A).

The standard calls for five reviews and two audits. These are: system requirements review, system design review, preliminary design review, critical design review, formal qualifications review, functional configuration audit, and physical configuration audit [20].

*Software Quality Assurance Program Requirements* (MIL-S-52779A).

The standard requires that the contractor develop and implement a Quality Assurance Program specifically for software. To accomplish this, the program must provide for detection, reporting, analysis, and correction of software deficiencies [25].

*Weapon System Software Development* (DOD-STD-1679A).

This standard address the complete software development process. It includes the following areas: software performance requirements; software design requirements; programming standards and conventions; software implementation; program regeneration; testing; software operation; software quality assurance; software acceptance; software configuration management; and software management planning [22].

*Tactical Digital System Standard. Software Quality Assurance Testing Criteria* (TADSTRAND 9).

The key requirements addressed are software endurance runs, third party conduct of endurance runs which are part of acceptance, allowable software errors, and allowable patches. Endurance runs include stress loading, degrading modes, and on-line maintenance support programs [28].

*Configuration Control—Engineering Changes, Deviations, and Waivers* (DOD-STD-480A).

This standard incorporates specific instructions for preparing software engineering change proposals [5].

*Military Standard Defense Systems Software Development* (draft of proposed MIL-STD-SDS).

The standard requires contractors to have software project management which provides an acceptable level of visibility into the development process and additional requirements for: structured and modular software architecture, requirements analysis, approved high order language, software integration testing, configuration management, software quality assurance, project planning and control, and subcontractor control [19]. This proposed standard began its approval process in 1982. Recommended changes from the Joint Logistics Commanders to MIL-STD-490, MIL-STD-483, and MIL-STD-1521A are included to make them compatible with MIL-STD-SDS.

Much of the material on military standards comes from [32] and [4].

73

## REFERENCES

- [1] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, Dec. 1976.
- [2] —, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
- [3] M. A. Branstad and P. B. Powell, "Process standards for software engineering," in *Proc. Software Eng. Standards Application Workshop*, Aug. 1981, IEEE Comput. Soc. Press.
- [4] J. D. Cooper, CACI Inc., Arlington, VA, lecture notes and conversions.
- [5] *Configuration Control—Engineering Changes, Deviations, and Waivers*, DOD-STD-480A, 1978.
- [6] *Guidelines for Documentation of Computer Programs and Automated Data Systems*, FIPS PUB 38, Feb. 1976.
- [7] *Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase*, FIPS PUB 64, Aug. 1979.
- [8] *Guideline: A Framework for the Evaluation and Comparison of Software Development Tools*, FIPS PUB 99, Mar. 1983.
- [9] *Guideline for Lifecycle Validation, Verification, and Testing of Computer Software*, FIPS PUB 101, June 1983.
- [10] G. N. Postel, "Principles of software standardization," in *Proc. Software Eng. Standards Application Workshop*, Aug. 1981, IEEE Comput. Soc. Press.
- [11] *IEEE Standard for Software Configuration Management Plans*, IEEE Std. 828-1983, IEEE Comput. Soc., New York, NY, 1983.
- [12] *IEEE Standard for Software Test Documentation*, ANSI/IEEE Std. 829-1983, IEEE Comput. Soc., New York, NY, 1983.
- [13] *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std. 729-1983, IEEE Comput. Soc., New York, NY, 1983.
- [14] *IEEE Standard for Software Quality Assurance Plans*, ANSI/IEEE Std. 730-1981, IEEE Comput. Soc., New York, NY, 1981.
- [15] *IEEE Guide for Software Requirements Specifications*, ANSI/IEEE Std. 830-1983, IEEE Comput. Soc., New York, NY, 1983.
- [16] B. M. Knight, "Organizational planning for software quality," in *Software Quality Management*. Petrocelli Books, 1979.
- [17] B. P. Lieritz and E. B. Swanson, *Software Maintenance Management*. Reading, MA: Addison-Wesley, 1980.
- [18] J. McCall et al., *Factors in Software Quality: Concept, and Definitions of Software Quality*, vol. 1, Nat. Tech. Inform. Service, AD/A-019 014, Nov. 1977.
- [19] *Military Standard Defense System Software Development*, Proposed MIL-STD-883, Joint Logistics Commanders, Apr. 1982.
- [20] *Technical Reviews and Audits for Systems, Equipment, and Computer Programs*, MIL-STD-1521A, USAF, Sept. 1978.
- [21] *Weapons Systems Software Development*, DOD-STD-1679A, Navy, Feb. 1983.
- [22] *Configuration Control—Engineering Changes, Deviations, and Waivers*, DOD-STD-480A, 1978.
- [23] *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs*, MIL-STD-483a, USAF, Dec. 1970.
- [24] *Specification Practices*, MIL-STD-490A, Oct. 1968.
- [25] *Software Quality Assurance Program Requirement*, MIL-STD-52779A, Army, 1977.
- [26] D. T. Ross, "Homilies for humble standards," *Commun. Ass. Comput. Mach.*, vol. 19, Nov. 1976.
- [27] *Software Engineering Standards Application Workshop*, Aug. 1981, Comput. Soc. Press.
- [28] *Standard Tactical Digital System Software Quality Assurance Testing Criteria*, TADSTAND 9, Navy, 1978.
- [29] L. L. Tripp, "Top-down, bottom-up approach to software engineering standards," in *Proc. Software Eng. Standards Application Workshop*, Aug. 1981, IEEE Comput. Soc. Press.
- [30] R. L. Van Tilburg, "Process standardization versus product standardization," *Software Eng. Standards* 1981.
- [31] G. H. Wedberg, "Implementation of software engineering standards," in *Proc. Software Eng. Standards Application Workshop*, Aug. 1981, IEEE Comput. Soc. Press.
- [32] D. L. Wood, "Department of Defense software quality requirements," in *Software Quality Management*. Petrocelli Books, 1979.



Martha Branstad received the B.S. degree in mathematics and the Ph.D. degree from Iowa State University, Ames, and the M.S. degree in mathematics from the University of Wisconsin.

She is Manager of the Software Engineering Group at the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC. She has primary responsibility for developing software engineering standards and guidelines which foster higher quality software within the Federal government. Her group

provides consulting services on software engineering techniques and technology to other agencies within the Federal government. Research in automated support to program construction and knowledge-based software development workstations is under her direction. Prior to joining NBS, she was the manager of a software research group at the National Security Agency. She has focused on natural language interfaces to databases, knowledge-based systems, software engineering, software development tools, and computer security. Previously she was an Assistant Professor of Computer Science at Iowa State University.



Patricia B. Powell received the B.A. degree from Smith College, Northampton, MA, and the M.S. degree in computer science from the University of Maryland, College Park.

She is a Computer Scientist at the Institute for Computer Sciences and Technology within the National Bureau of Standards, Washington, DC. Her current responsibilities are in the areas of technology forecasting and cost-benefit analysis. She also participates in IEEE standards working groups. Previously, she worked in the areas of software engineering, artificial intelligence, and the development of high-order languages.

Mrs. Powell is a member of the IEEE Computer Society and the Association for Computing Machinery.

# Elements of Software Configuration Management

EDWARD H. BERSOFF, SENIOR MEMBER, IEEE

74

## SCM IN CONTEXT

**Abstract**—Software configuration management (SCM) is one of the disciplines of the 1980's which grew in response to the many failures of the software industry throughout the 1970's. Over the last ten years, computers have been applied to the solution of so many complex problems that our ability to manage these applications has all too frequently failed. This has resulted in the development of a series of "new" disciplines intended to help control the software process.

This paper will focus on the discipline of SCM by first placing it in its proper context with respect to the rest of the software development process, as well as to the goals of that process. It will examine the constituent components of SCM, dwelling at some length on one of those components, configuration control. It will conclude with a look at what the 1980's might have in store.

**Index Terms**—Configuration management, management, product assurance, software.

## INTRODUCTION

SOFTWARE configuration management (SCM) is one of the disciplines of the 1980's which grew in response to the many failures of our industry throughout the 1970's. Over the last ten years, computers have been applied to the solution of so many complex problems that our ability to manage these applications in the "traditional" way has all too frequently failed. Of course, tradition in the software business began only 30 years ago or less, but even new habits are difficult to break. In the 1970's we learned the hard way that the tasks involved in managing a software project were not linearly dependent on the number of lines of code produced. The relationship was, in fact, highly exponential. As the decade closed, we looked back on our failures [1], [2] trying to understand what went wrong and how we could correct it. We began to dissect the software development process [3], [4] and to define techniques by which it could be effectively managed [5]-[8]. This self-examination by some of the most talented and experienced members of the software community led to the development of a series of "new" disciplines intended to help control the software process.

While this paper will focus on the particular discipline of SCM, we will first place it in its proper context with respect to the rest of the software development process, as well as to the goals of that process. We will examine the constituent components of SCM, dwelling at some length on one of those components, configuration control. Once we have woven our way through all the trees, we will once again stand back and take a brief look at the forest and see what the 1980's might have in store.

Manuscript received April 15, 1982; revised December 1, 1982 and October 18, 1983.

The author is with BTG, Inc., 1945 Gallows Rd., Vienna, VA 22180.

It has been said that if you do not know where you are going, any road will get you there. In order to properly understand the role that SCM plays in the software development process, we must first understand what the goal of that process is, i.e., where we are going. For now, and perhaps for some time to come, software developers are people, people who respond to the needs of another set of people creating computer programs designed to satisfy those needs. These computer programs are the tangible output of a thought process—the conversion of a thought process into a product. The goal of the software developer is, or should be, the construction of a product which closely matches the real needs of the set of people for whom the software is developed. We call this goal the achievement of "product integrity." More formally stated, product integrity (depicted in Fig. 1) is defined to be the intrinsic set of attributes that characterize a product [9]:

- that fulfills user functional needs;
- that can easily and completely be traced through its life cycle;
- that meets specified performance criteria;
- whose cost expectations are met;
- whose delivery expectations are met.

The above definition is pragmatically based. It demands that product integrity be a measure of the satisfaction of the real needs and expectations of the software user. It places the burden for achieving the software goal, product integrity, squarely on the shoulders of the developer, for it is he alone who is in control of the development process. While, as we shall see, the user can establish safeguards and checkpoints to gain visibility into the development process, the prime responsibility for software success is the developer's. So our goal is now clear; we want to build software which exhibits all the characteristics of product integrity. Let us make sure that we all understand, however, what this thing called software really is. We have learned in recent times that equating the terms "software" and "computer programs" improperly restricts our view of software. Software is much more. A definition which can be used to focus the discussion in this paper is that software is information that is:

- structured with logical and functional properties;
- created and maintained in various forms and representations during the life cycle;
- tailored for machine processing in its fully developed state.

So by our definition, software is not simply a set of computer programs, but includes the documentation required to define, develop, and maintain these programs. While this notion is not very new, it still frequently escapes the software

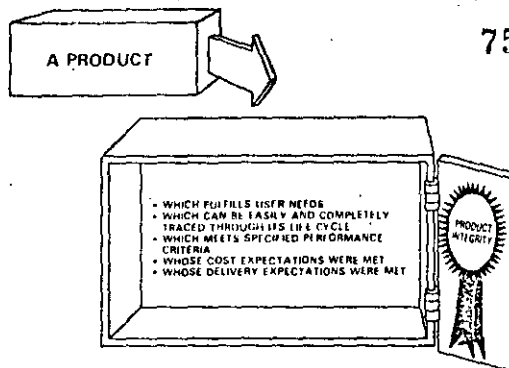


Fig. 1. Product integrity.

development manager who assumes that controlling a software product is the same as controlling computer code.

Now that we more fully appreciate what we are after, i.e., to build a software product with integrity, let us look at the one road which might get us there. We have, until now, used the term "developer" to characterize the organizational unit responsible for converting the software idea into a software product. But developers are, in reality, a complex set of interacting organizational entities. When undertaking a software project, most developers structure themselves into three basic discipline sets which include:

- project management,
- development, and
- product assurance.

Project management disciplines are both inwardly and outwardly directed. They support general management's need to see what is going on in a project and to ensure that the parent or host organization consistently develops products with integrity. At the same time, these disciplines look inside a project in support of the assignment, allocation, and control of all project resources. In that capacity, project management determines the relative allocation of resources to the set of development and product assurance disciplines. It is management's prerogative to specify the extent to which a given discipline will be applied to a given project. Historically, management has often been handicapped when it came to deciding how much of the product assurance disciplines were required. This was a result of both inexperience and organizational immaturity.

The development disciplines represent those traditionally applied to a software project. They include:

- analysis,
- design,
- engineering,
- production (coding),
- test (unit/subsystem),
- installation,
- documentation,
- training, and
- maintenance.

In the broadest sense, these are the disciplines required to take a system concept from its beginning through the development life cycle. It takes a well-structured, rigorous technical approach to system development, along with the right mix of development disciplines to attain product integrity, especially for software. The concept of an ordered, procedurally disciplined approach to system development is fundamental to product integrity. Such an approach provides successive development plateaus, each of which is an identifiable measure of progress which forms a part of the total foundation supporting the final product. Going sequentially from one baseline (plateau) to another with high probability of success, necessitates the use of the right development disciplines at precisely the right time.

The product assurance disciplines which are used by project management to gain visibility into the development process include:

- configuration management,
- quality assurance,
- validation and verification, and
- test and evaluation.

Proper employment of these product assurance disciplines by the project manager is basic to the success of a project since they provide the technical checks and balances over the product being developed. Fig. 2 represents the relationship among the management, development, and product assurance disciplines. Let us look at each of the product assurance disciplines briefly, in turn, before we explore the details of SCM.

Configuration management (CM) is the discipline of identifying the configuration of a system at discrete points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle. Software configuration management (SCM) is simply configuration management tailored to systems, or portions of systems, that are comprised predominantly of software. Thus, SCM does not differ substantially from the CM of hardware-oriented systems, which is generally well understood and effectively practiced. However, attempts to implement SCM have often failed because the particulars of SCM do not follow by direct analogy from the particulars of hardware CM and because SCM is a less mature discipline than that of hardware CM. We will return to this subject shortly.

Quality assurance (QA) as a discipline is commonly invoked throughout government and industry organizations with reasonable standardization when applied to systems comprised only of hardware. But there is enormous variation in thinking and practice when the QA discipline is invoked for a software development or for a system containing software components. QA has a long history, and much like CM, it has been largely developed and practiced on hardware projects. It is therefore mature, in that sense, as a discipline. Like CM, however, it is relatively immature when applied to software development. We define QA as consisting of the procedures, techniques, and tools applied by professionals to insure that a product meets or exceeds prespecified standards during a product's development cycle; and without specific prescribed standards, QA entails insuring that a product meets or

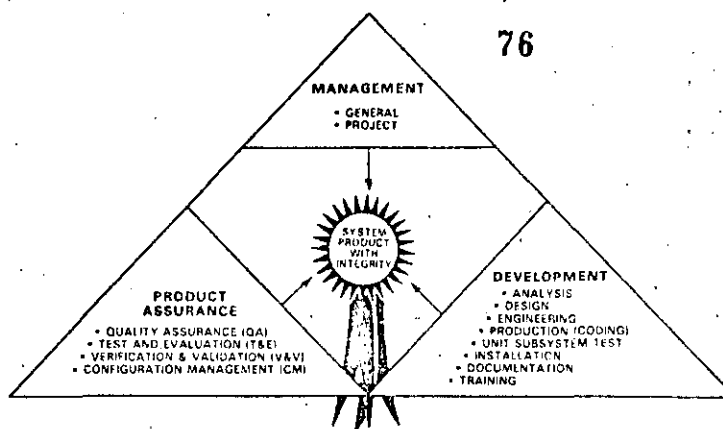


Fig. 2. The discipline triangle.

exceeds a minimum industrial and/or commercially acceptable level of excellence.

The QA discipline has not been uniformly treated, practiced or invoked relative to software development. First, very few organizations have software design and development standards that compare in any way with hardware standards for detail and completeness. Second, it takes a high level of software expertise to assess whether a software product meets prescribed standards. Third, few buyer organizations have provided for or have developed the capability to impose and then monitor software QA endeavors on seller organizations. Finally, few organizations have been concerned over precisely defining the difference between QA and other product assurance disciplines, CM often being subservient to QA or vice versa in a given development organization. Our definition of software QA discipline being in the same state as SCM so far as its universal application within the user, buyer, and seller communities. Software, as a form of information, cannot be standardized; only structures for defining/documenting software can be standardized. It follows that software development techniques can only be meaningfully standardized in relation to information structures, not information content.

The third of the four product assurance disciplines is validation and verification (V&V). Unlike CM and QA, V&V has come into being expressly for the purpose of coping with software and its development. Unlike QA, which principally deals with the problem of a product's adherence to pre-established standards, V&V deals with the issue of how well software fulfills functional and performance requirements and the assurance that specified requirements are indeed stated and interpreted correctly. The verification part of V&V assures that a product meets its prescribed goals as defined through baseline documentation. That is, verification is a discipline imposed to ascertain that a product is what it was intended to be relative to its preceding baseline. The validation part of V&V, by contrast, is levied as a discipline to assure that a product not only meets the objectives specified through baseline documentation, but in addition, does the right job.

Stated another way, the validation discipline is invoked to insure that the end-user gets the right product. A buyer or seller may have misinterpreted user requirements or, perhaps, requirements have changed, or the user gets to know more about what he needs, or early specifications of requirements were wrong or incomplete or in a state of flux. The validation process serves to assure that such problems do not persist among the user, buyer, and seller. To enhance objectivity, it is often desirable to have an independent organization, from outside the developing organization, perform the V&V function.

The fourth of the product assurance disciplines is test and evaluation (T&E), perhaps the discipline most understood, and yet paradoxically, least practiced with uniformity. T&E is defined as the discipline imposed outside the development project organization to independently assess whether a product fulfills objectives. T&E does this through the execution of a set of test plans and procedures. Specifically in support of the end user, T&E entails evaluating product performance in a live or near-live environment. Frequently, particularly within the military arena, T&E is a major undertaking involving one or more systems which are to operate together, but which have been individually developed and accepted as stand-alone items. Some organizations formally turn over T&E responsibility to a group outside the development project organization after the product reaches a certain stage of development, their philosophy being that developers cannot be objective to the point of fully testing/evaluating what they have produced.

The definitions given for CM, QA, V&V, and T&E suggest some overlap in required skills and functions to be performed in order to invoke these disciplines collectively for product assurance purposes. Depending on many factors, the actual overlap may be significant or little. In fact, there are those who would argue that V&V and T&E are but subset functions of QA. But the contesting argument is that V&V and T&E have come into being as separate disciplines because conventional QA methods and techniques have failed to do an adequate job with respect to providing product assurance, par-

ticularly for computer-centered systems with software components. Management must be concerned with minimizing the application of excessive and redundant resources to address the overlap of these disciplines. What is important is that all the functions defined above are performed, not what they are called or who carries them out.

77

#### THE ELEMENTS OF SCM

When the need for the discipline of configuration management finally achieved widespread recognition within the software engineering community, the question arose as to how closely the software CM discipline ought to parallel the extant hardware practice of configuration management. Early SCM authors and practitioners [10] wisely chose the path of commonality with the hardware world, at least at the highest level. Of course, hardware engineering is different from software engineering, but broad similarities do exist and terms applied to one segment of the engineering community can easily be applied to another, even if the specific meanings of those terms differ significantly in detail. For that reason, the elements of SCM were chosen to be the same as those for hardware CM. As for hardware, the four components of SCM are:

- identification,
- control,
- auditing, and
- status accounting.

Let us examine each one in turn.

*Software Configuration Identification:* Effective management of the development of a system requires careful definition of its baseline components; changes to these components also need to be defined since these changes, together with the baselines, specify the system evolution. A system baseline is like a snapshot of the aggregate of system components as they exist at a given point in time; updates to this baseline are like frames in a movie strip of the system life cycle. The role of software configuration identification in the SCM process is to provide labels for these snapshots and the movie strip.

A baseline can be characterized by two labels. One label identifies the baseline itself, while the second label identifies an update to a particular baseline. An update to a baseline represents a baseline plus a set of changes that have been incorporated into it. Each of the baselines established during a software system's life cycle controls subsequent system development. At the time it is first established a software baseline embodies the actual software in its most recent state. When changes are made to the most recently established baseline, then, from the viewpoint of the software configuration manager, this baseline and these changes embody the actual software in its most recent state (although, from the viewpoint of the software developer, the actual software may be in a more advanced state).

The most elementary entity in the software configuration identification labeling mechanism is the software configuration item (SCI). Viewed from an SCM perspective, a software baseline appears as a set of SCI's. The SCI's within a baseline are related to one another via a tree-like hierarchy. As the software system evolves through its life cycle, the number of

branches in this hierarchy generally increases; the first baseline may consist of no more than one SCI. The lowest level SCI's in the tree hierarchy may still be under development and not yet under SCM control. These entities are termed design objects or computer program components (see Fig. 3). Each baseline and each member in the associated family of updates will exist in one or more forms, such as a design document, source code on a disk, or executing object code.

In performing the identification function, the software configuration manager is, in effect, taking snapshots of the SCI's. Each baseline and its associated updates collectively represents the evolution of the software during each of its life cycle stages. These stages are staggered with respect to one another. Thus, the collection of life cycle stages looks like a collection of staggered and overlapping sequences of snapshots of SCI trees. Let us now imagine that this collection of snapshot sequences is threaded, in chronological order, onto a strip of movie film as in Fig. 4. Let us further imagine that the strip of movie film is run through a projector. Then we would see a history of the evolution of the software. Consequently, the identification of baselines and updates provides an explicit documentation trail linking all stages of the software life cycle. With the aid of this documentation trail, the software developer can assess the integrity of his product, and the software buyer can assess the integrity of the product he is paying for.

*Software Configuration Control:* The evolution of a software system is, in the language of SCM, the development of baselines and the incorporation of a series of changes into the baselines. In addition to these changes that explicitly affect existing baselines, there are changes that occur during early stages of the system life cycle that may affect baselines that do not yet exist. For example, some time before software coding begins (i.e., some time prior to the establishment of a design baseline), a contract may be modified to include a software warranty provision such as: system downtime due to software failures shall not exceed 30 minutes per day. This warranty provision will generally affect subsequent baselines but in a manner that cannot be explicitly determined *a priori*. One role of software configuration control is to provide the administrative mechanism for precipitating, preparing, evaluating, and approving or disapproving all change proposals throughout the system life cycle.

We have said that software, for configuration management purposes, is a collection of SCI's that are related to one another in a well-defined way. In early baselines and their associated updates, SCI's are specification documents (one or more volumes of text for each baseline or associated update); in later baselines and their associated updates, each SCI may manifest itself in any or all of the various software representations. Software configuration control focuses on managing changes to SCI's (existing or to be developed) in all of their representations. This process involves three basic ingredients.

- 1) Documentation (such as administrative forms and supporting technical and administrative material) for formally precipitating and defining a proposed change to a software system.
- 2) An organizational body for formally evaluating and

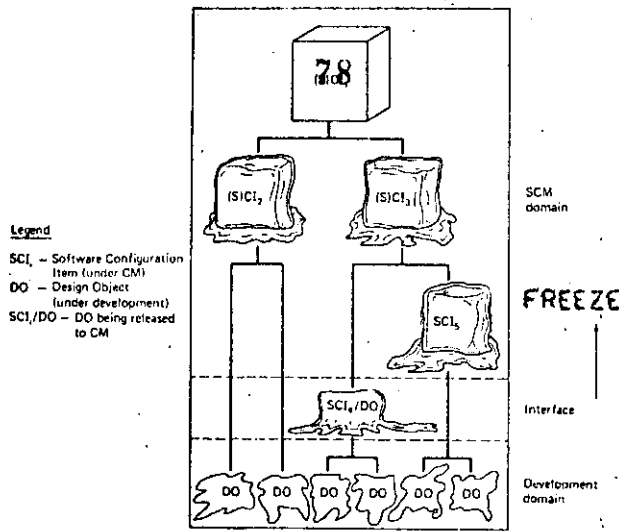


Fig. 3. The development/SCM interface.

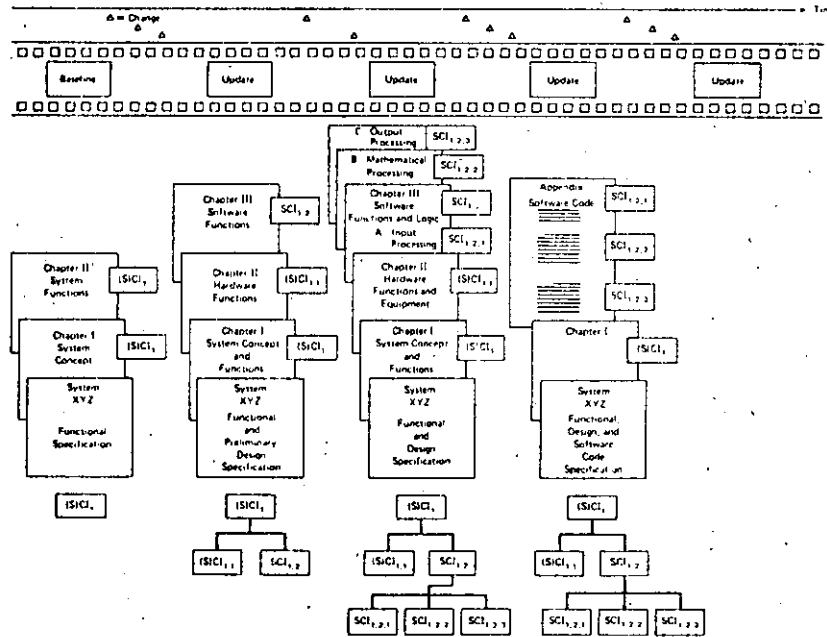


Fig. 4. SCL evolution in a single document.

approving or disapproving a proposed change to a software system (the Configuration Control Board).

3) Procedures for controlling changes to a software system. The Engineering Change Proposal (ECP), a major control document, contains information such as a description of the proposed change, identification of the originating organization,

rationale for the change, identification of affected baselines and SCL's (if appropriate), and specification of cost and schedule impacts. ECP's are reviewed and coordinated by the CCB, which is typically a body representing all organizational units which have a vested interest in proposed changes.

Fig. 5 depicts the software configuration control process.



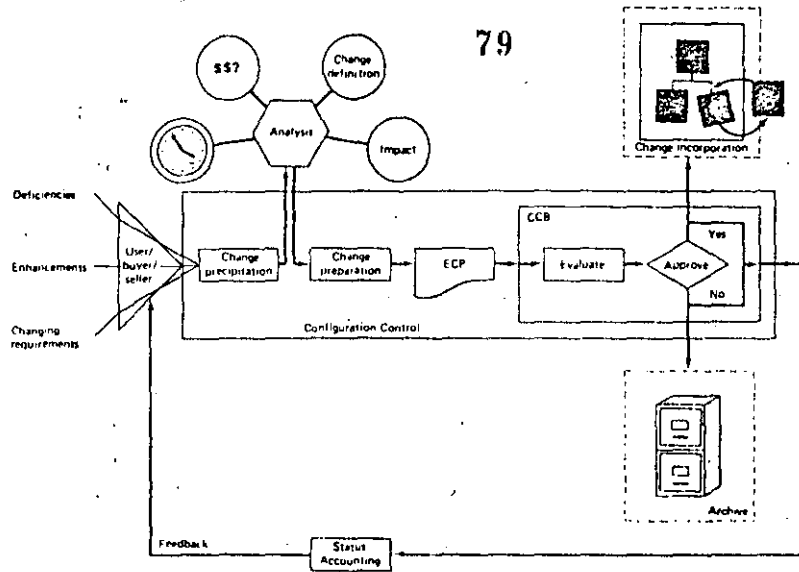


Fig. 5. The control process.

As the figure suggests, change incorporation is not an SCM function, but monitoring the change implementation process resulting in change incorporation is. Fig. 5 also emphasizes that the analysis that may be required to prepare an ECP is also outside the SCM purview. Note also from the figure how ECP's not approved by the CCB are not simply discarded but are archived for possible future reference.

Many automated tools support the control process. The major ones aid in controlling software change once the coding stage has been reached, and are generically referred to as program support libraries (PSL's). The level of support provided by PSL's, however, varies greatly. As a minimum, a PSL should provide a centralized and readily available repository for authoritative versions of each component of a software system. It should contain the data necessary for the orderly development and control of each SCI. Automation of other functions, such as library access control, software and document version maintenance, change recording, and document reconstruction, greatly enhance both the control and maintenance processes. These capabilities are currently available in systems such as SOFTOOL's change and configuration control environment (CCC).

A PSL supports a developmental approach in which project personnel work on a common visible product rather than on independent components. In those PSL's which include access controls, project personnel can be separately assigned read/write access to each software document/component, from programs to lines of code. Thus, all project personnel are assured ready access to the critical interface information necessary for effective software development. At the same time, modifications to various software components, whether sanctioned baselines or modules under development, can be closely controlled.

Under the PSL concept, the programmer operates under a well-defined set of parameters and exercises a narrower span

of detailed control. This minimizes the need for explicit communication between analysts and programmers and makes the inclusion of new project personnel less traumatic since interface requirements are well documented. It also minimizes the preparation effort for technical audits.

Responsibility for maintenance of the PSL data varies depending on the level of automation provided. For those systems which provide only a repository for data, a secretary/librarian is usually responsible for maintaining the notebooks which will contain the data developed and used by project personnel and for maintenance of the PSL archives. More advanced PSL systems provide real time, on-line access to data and programs and automatically create the records necessary to fully trace the history of the development. In either case the PSL provides standardization of project recordkeeping, ensures that system documentation corresponds to the current system configuration, and guarantees the existence of adequate documentation of previous versions.

A PSL should support three main activities: code development, software management, and configuration control. Support to the development process includes support to design, coding, testing, documentation, and program maintenance along with associated database schema and subschema. A PSL provides this support through:

- storage and maintenance of software documentation and code,
- support to program compilation/testing,
- support for the generation of program/system documentation.

Support to the management of the software development process involves the storage and output of programming data such as:

- collection and automatic reporting of management data related to program development,

- control over the integrity and security of the data in the PSL,
- separation of the clerical activity related to the programming process.

80

PSL's provide support to the configuration control process through:

- access and change authorization control for all data in the library,
- control of software code releases,
- automatic program and document reconstruction,
- automatic change tracking and reporting,
- assurance of the consistency between documentation, code, and listings.

A PSL has four major components: internal libraries in machine-readable form, external libraries in hardcopy form, computer procedures, and office procedures. The components of a PSL system are interlocked to establish an exact correspondence between the internal units of code and external versions (such as listings) of the developing systems. This continuous correspondence is the characteristic of a PSL that guarantees ongoing visibility and identification of the developing system.

Different PSL implementations exist for various system environments with the specifics of the implementation dependent upon the hardware, software, user, and operating environment. The fundamental correspondence between the internal and external libraries in each environment, however, is established by the PSL librarian and computer procedures. The office procedures are specified in a project CM Plan so that the format of the external libraries is standard across software projects, and internal and external libraries are easily maintainable.

Newer PSL systems minimize the need for both office and computer procedures through the implementation of extensive management functionality. This functionality provides significant flexibility in controlling the access to data and allocating change authority, while providing a variety of status reporting capabilities. The availability of management information, such as a list of all the software structures changed to solve a particular Software Trouble Report or the details on the latest changes to a particular software document, provides a means for the control function to effectively operate without burdening the development team with cumbersome procedures and administrative paperwork. Current efforts in PSL refinement/development are aimed at linking support of the development environment with that of the configuration control environment. The goal of such systems is to provide an integrated environment where control and management information is generated automatically as a part of a fully supported design and development process.

*Software Configuration Auditing:* Software configuration auditing provides the mechanism for determining the degree to which the current state of the software system mirrors the software system pictured in baseline and requirements documentation. It also provides the mechanism for formally establishing a baseline. A baseline in its formative stages (for example, a draft specification document that appears prior to the existence of the functional baseline) is referred to as a "to-be-established" baseline; the final state of the auditing process

conducted on a to-be-established baseline is a sanctioned baseline. The same may be said about baseline updates.

Software configuration auditing serves two purposes, configuration verification and configuration validation. Verification ensures that what is intended for each software configuration item as specified in one baseline or update is actually achieved in the succeeding baseline or update; validation ensures that the SCI configuration solves the right problem (i.e., that customer needs are satisfied). Software configuration auditing is applied to each baseline (and corresponding update) in its to-be-established state. An auditing process common to all baselines is the determination that an SCI structure exists and that its contents are based on all available information.

Software auditing is intended to increase software visibility and to establish traceability throughout the life cycle of the software product. Of course, this visibility and traceability are not achieved without cost. Software auditing costs time and money. But the judicious investment of time and money, particularly in the early stages of a project, pays dividends in the latter stages. These dividends include the avoidance of costly retrofits resulting from problems such as the sudden appearance of new requirements and the discovery of major design flaws. Conversely, failing to perform auditing, or constraining it to the later stages of the software life cycle, can jeopardize successful software development. Often in such cases, by the time discrepancies are discovered (if they are), the software cannot be easily or economically modified to rectify the discrepancies. The result is often a dissatisfied customer, large cost overruns, slipped schedules, or cancelled projects.

Software auditing makes visible to management the current status of the software in the life cycle product audited. It also reveals whether the project requirements are being satisfied and whether the intent of the preceding baseline has been fulfilled. With this visibility, project management can evaluate the integrity of the software product being developed, resolve issues that may have been raised by the audit, and correct defects in the development process. The visibility afforded by the software audit also provides a basis for the establishment of the audited life cycle product as a new baseline.

Software auditing provides traceability between a software life cycle product and the requirements for that product. Thus, as life cycle products are audited and baselines established, every requirement is traced successively from baseline to baseline. Disconnects are also made visible during the establishment of traceability. These disconnects include requirements not satisfied in the audited product and extraneous features observed in the product (i.e., features for which no stated requirement exists).

With the different point of view made possible by the visibility and traceability achieved in the software audit, management can make better decisions and exercise more incisive control over the software development process. The result of a software audit may be the establishment of a baseline, the redirection of project tasking, or an adjustment of applied project resources.

The responsibility for a successful software development project is shared by the buyer, seller, and user. Software auditing uniquely benefits each of these project participants. Appropriate auditing by each party provides checks and

balances over the development effort. The scope and depth of the audits undertaken by the three parties may vary greatly. However, the purposes of these differing forms of software audit remain the same: to provide visibility and to establish traceability of the software life cycle products. An excellent overview of the software audit process, from which some of the above discussion has been extracted, appears in [11].

**Software Configuration Status Accounting:** A decision to make a change is generally followed by a time delay before the change is actually made, and changes to baselines generally occur over a protracted period of time before they are incorporated into baselines as updates. A mechanism is therefore needed for maintaining a record of how the system has evolved and where the system is at any time relative to what appears in published baseline documentation and written agreements. Software configuration status accounting provides this mechanism. Status accounting is the administrative tracking and reporting of all software items formally identified and controlled. It also involves the maintenance of records, to support software configuration auditing. Thus, software configuration status accounting records the activity associated with the other three SCM functions and therefore provides the means by which the history of the software system life cycle can be traced.

Although administrative in nature, status accounting is a function that increases in complexity as the system life cycle progresses because of the multiple software representations that emerge with later baselines. This complexity generally results in large amounts of data to be recorded and reported. In particular, the scope of software configuration status accounting encompasses the recording and reporting of:

- 1) the time at which each representation of a baseline and update came into being;
- 2) the time at which each software configuration item came into being;
- 3) descriptive information about each SCI;
- 4) engineering change proposal status (approved, disapproved, awaiting action);
- 5) descriptive information about each ECP;
- 6) change status;
- 7) descriptive information about each change;
- 8) status of technical and administrative documentation associated with a baseline or update (such as a plan prescribing tests to be performed on a baseline for updating purposes);
- 9) deficiencies in a to-be-established baseline uncovered during a configuration audit.

Software configuration status accounting, because of its large data input and output requirements, is generally supported in part by automated processes such as the PSL described earlier. Data are collected and organized for input to a computer and reports giving the status of entities are compiled and generated by the computer.

#### THE MANAGEMENT DILEMMA

As we mentioned at the beginning of this paper, SCM and many of the other product assurance disciplines grew up in the 1970's in response to software failure. The new disciplines were designed to achieve visibility into the soft-

ware engineering process and thereby exercise some measure of control over that process. Students of mathematical control theory are taught early in their studies a simple example of the control process. Consider being confronted with a cup of hot coffee, filled to the top, which you are expected to carry from the kitchen counter to the kitchen table. It is easily verified that if you watch the cup as you carry it, you are likely to spill more coffee than if you were to keep your head turned away from the cup. The problem with looking at the cup is one of overcompensation. As you observe slight deviations from the straight-and-level, you adjust, but often you adjust too much. To compensate for that overadjustment, you tend to overadjust again, with the result being hot coffee on your floor.

This little diversion from our main topic of SCM has an obvious moral. There is a fundamental propensity on the part of the practitioners of the product assurance disciplines to overadjust, to overcompensate for the failures of the development disciplines. There is one sure way to eliminate failure completely from the software development process, and that is to stop it completely. The software project manager must learn how to apply his resources intelligently. He must achieve visibility and control, but he must not so encumber the developer so as to bring progress to a virtual halt. The product assurers have a virtuous perspective. They strive for perfection and point out when and where perfection has not been achieved. We seem to have a binary attitude about software; it is either correct or it is not. That is perhaps true, but we cannot expect anyone to deliver perfect software in any reasonable time period or for a reasonable sum of money. What we need to develop is software that is good enough. Some of the controls that we have placed on the developer have the deleterious effect, of increasing costs and expanding schedules rather than shrinking them.

The dilemma to management is real. We must have the visibility and control that the product assurance disciplines have the capacity to provide. But we must be careful not to overcompensate and overcontrol. This is the fine line which will distinguish the successful software managers of the 1980's from the rest of the software engineering community.

#### ACKNOWLEDGMENT

The author wishes to acknowledge the contribution of B. J. Gregor to the preparation and critique of the final manuscript.

#### REFERENCES

- [1] "Contracting for computer software development—Serious problems require management attention to avoid wasting additional millions," General Accounting Office, Rep. FGMSD 80-4, Nov. 9, 1979.
- [2] D. M. Weiss, "The MUDD report: A case study of Navy software development practices," Naval Res. Lab., Rep. 7909, May 21, 1975.
- [3] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226-1241, Dec. 1976.
- [4] *Proc. IEEE* (Special Issue on Software Engineering), vol. 68, Sept. 1980.
- [5] E. Bersoff, V. Henderson, and S. Siegel, "Attaining software product integrity," *Tutorial: Software Configuration Management*, W. Bryan, C. Chadbourne, and S. Siegel, Eds., Los Alamitos, CA, IEEE Comput. Soc., Cat. E110-169-3, 1981.
- [6] B. W. Boehm et al., *Characteristics of Software Quality*, TRW Series of Software Technology, vol. 1. New York: North-Holland, 1978.
- [7] T. A. Thayer, et al., *Software Reliability*, TRW Series of Software Technology, vol. 2. New York: North-Holland, 1978.

- [8] D. J. Reifer, Ed., *Tutorial: Automated Tools for Software Eng.*, Los Alamitos, CA, IEEE Comput. Soc., Cat. EHO-169-3, 1979.
- [9] E. Bersoff, V. Henderson, and S. Siegel, *Software Configuration Management*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
- [10] —, "Software configuration management: A tutorial," *Computer*, vol. 12, pp. 6-14, Jan. 1979.
- [11] W. Bryan, S. Siegel, and G. Whiteleather, "Auditing throughout the software life cycle: A primer," *Computer*, vol. 15, pp. 56-67, Mar. 1982.
- [12] "Software configuration management," Naval Elec. Syst. Command, Software Management Guidebooks, vol. 2, undated.



Edward H. Bersoff (M'75-SM'78) received the A.B., M.S., and Ph.D. degrees in mathematics from New York University, New York.

He is President and Founder of BTG, Inc., a high technology, Washington, DC area based, systems analysis and engineering firm. In addition to his corporate responsibilities, he directs the company's research in software engineering, product assurance, and software management. BTG specializes in the application of modern systems engineering principles to the computer

based system development process. At BTG, he has been actively in-

involved in the FAA's Advanced Automation Program where he is focusing on software management and software configuration management issues on this extremely complex program. He also participates in the company's activities within the Naval Intelligence community, providing senior consulting services to a wide variety of system development efforts. He was previously President of CTEC, Inc. where he directed the concept formulation and development of the Navy Command and Control System (NCCS), Ocean Surveillance Information System (OSIS) Baseline now installed at all U.S. Navy Ocean Surveillance Centers. He also served as Experiment Director for the Joint ARPA, Navy, CINCPAC Military Message Experiment. This test was designed to examine the usefulness of secure, automated message processing systems in an operational military environment and to develop design criteria for future military message processing systems. Prior to joining CTEC, Inc., he was Manager of Engineering Operations and Manager of FAA Operations for Logicon, Inc.'s Process Systems Division. He joined Logicon from the NASA Electronics Research Center. He has taught mathematics at universities in Boston, New York, and Washington, DC. His technical contributions to the fields of software requirements and design range from early publications in computer architecture, reliability and programming languages, to more recent publications in software quality and configuration management. A textbook entitled *Software Configuration Management* (Prentice-Hall) represents the product of three years of research in the field by Dr. Bersoff and his colleagues.

Dr. Bersoff is a member of AFCEA, American Management Association, MENSA, and the Young Presidents' Organization.



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

S O F T W A R E

Ing. Daniel Ríos Zertuche

NOVIEMBRE, 1984

## Software

Programas de computadora y su documentación asociada, requerida para su desarrollo, operación, y mantenimiento.

## Ingeniería de Software

La aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollarlos, operarlos y mantenerlos.

Es la aplicación de la ciencia y matemáticas por medio de la cual la capacidad del equipo de cómputo se hacen útiles al hombre por medio de programas de computadora, procedimientos y documentación asociada.

Que es el Software?

El Software es información:

- 1). Estructurada con propiedades lógicas y funcionales.
- 2). Creada y mantenida en varias formas y representaciones durante su ciclo de vida.
- 3). Fabricado para una maquina en su estado de desarrollo completo.

Existe en 2 formas básicas:

No ejecutable

Documentación

procesable en máquina

Ejecutable

La Ingeniería de Software determina el costo y la calidad del Software producido.

---

### COSTO

El Software es caro y su costo tiende a ser mayor.

En 1980 en U.S.A. el gasto en Software fué de --  
40,000 millones de dólares 2% del PIB.

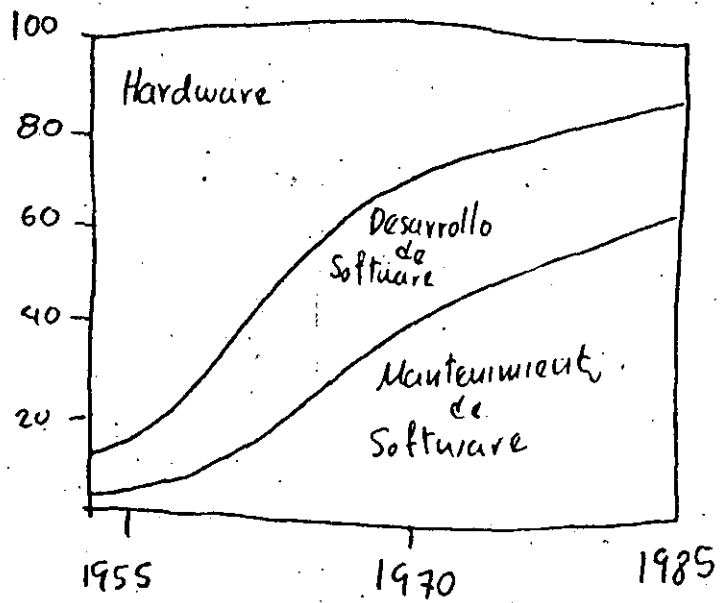
Para 1990 podría llegar a ser el 13% del PIB.

El reto aquí es de dos tipos:

- a) Incrementar significativamente la productividad del desarrollo de Software.
- b) Incrementar la eficiencia del mantenimiento del Software.

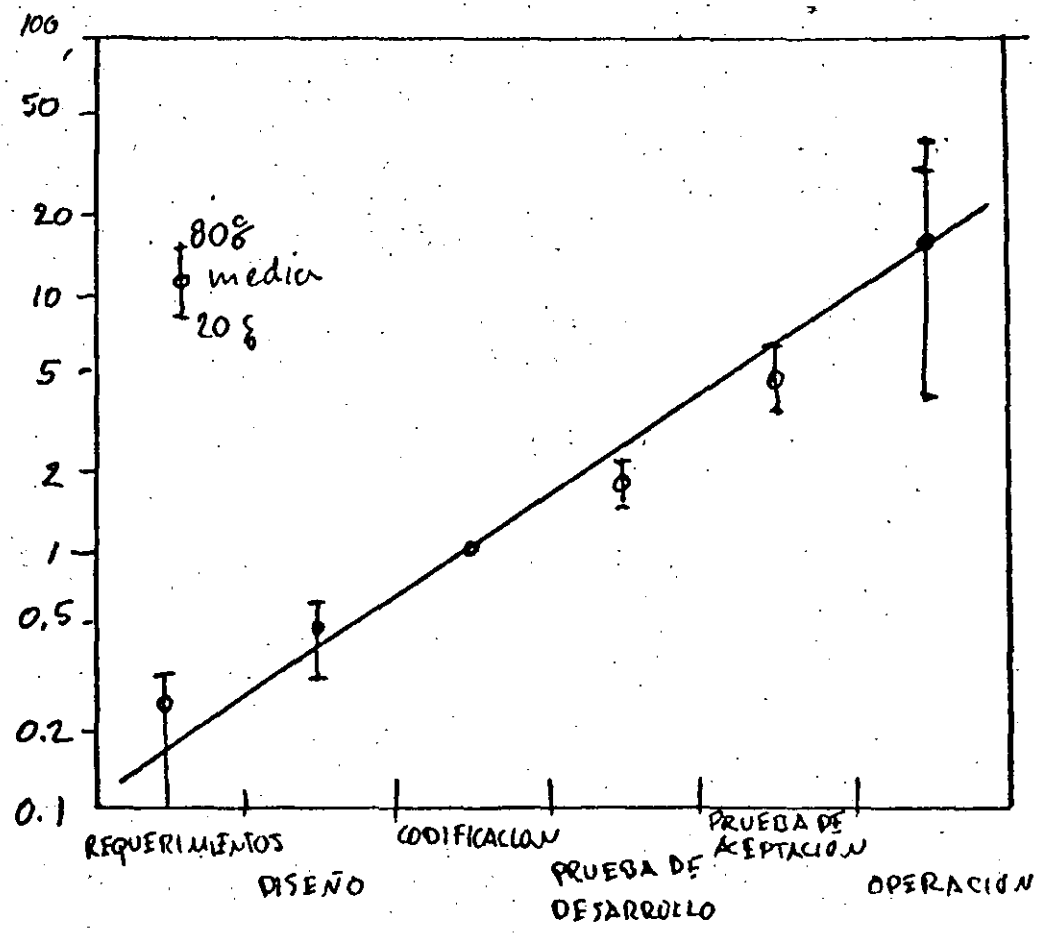


Porcentaje del costo



TENDENCIAS DEL COSTO HARDWARE-SOFTWARE

COSTO RELATIVO DE CORREGIR UN ERROR



FASE EN LA CUAL SE DETECTA EL ERROR

## IMPACTO SOCIAL

El papel que juega el Software dentro de la sociedad cada vez es más importante.

El crecimiento de la demanda de Software tiene su origen en el hecho que conforme el Hardware se hace más económico, confiable y poderoso, se encuentran mayores ventajas para automatizar las partes mecánicas de las tareas de los humanos.

El incremento en el impacto en el bienestar humano requiere que se desarrolle y mantenga Software que sea:

Extremadamente Confiable

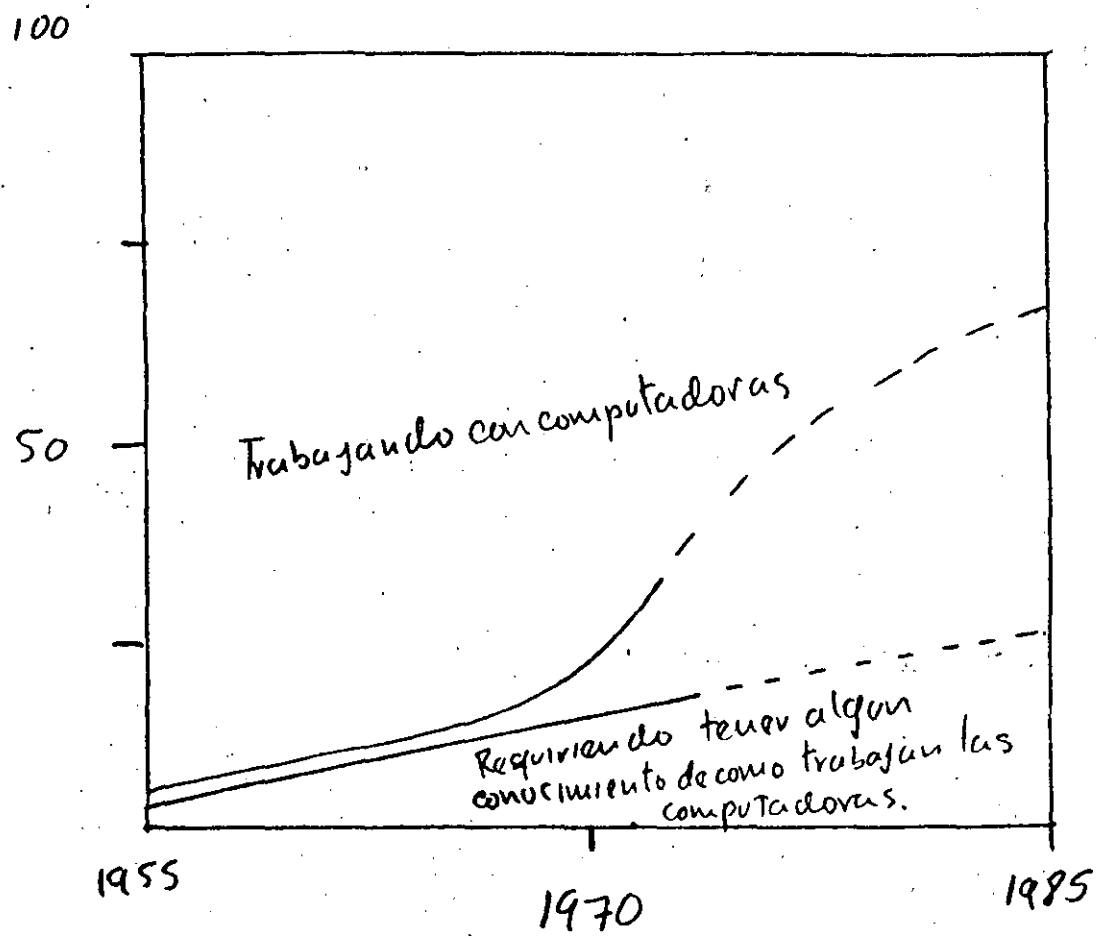
Humano

Fácil de usar

Difícil de usar mal

Auditable

Porcentaje de la fuerza de Trabajo



Crecimiento de la confianza en las computadoras y el software.

Enfoque Orientado a Metas para el Ciclo de Vida del Software

1. Defina las metas principales a ser alcanzadas por el producto de Software y el proceso de Software.
2. Use la estructura de metas de la Ingeniería de Software como una lista de puntos a checar, para asegurar que se han identificado todas las metas principales.
3. Defina los medios por los cuales se lograrán estas metas.

Esto incluye definir un plan para:

- . Quien es el responsable de lograr cada meta.
- . Cuando y donde se logrará cada meta.
- . Como se logrará cada meta. Esto incluye la definición de cualquier conjunto adicional de metas de submetas requeridas y su secuencia.
- . Que supuestos deben ser validos con el fin de alcanzar las metas.

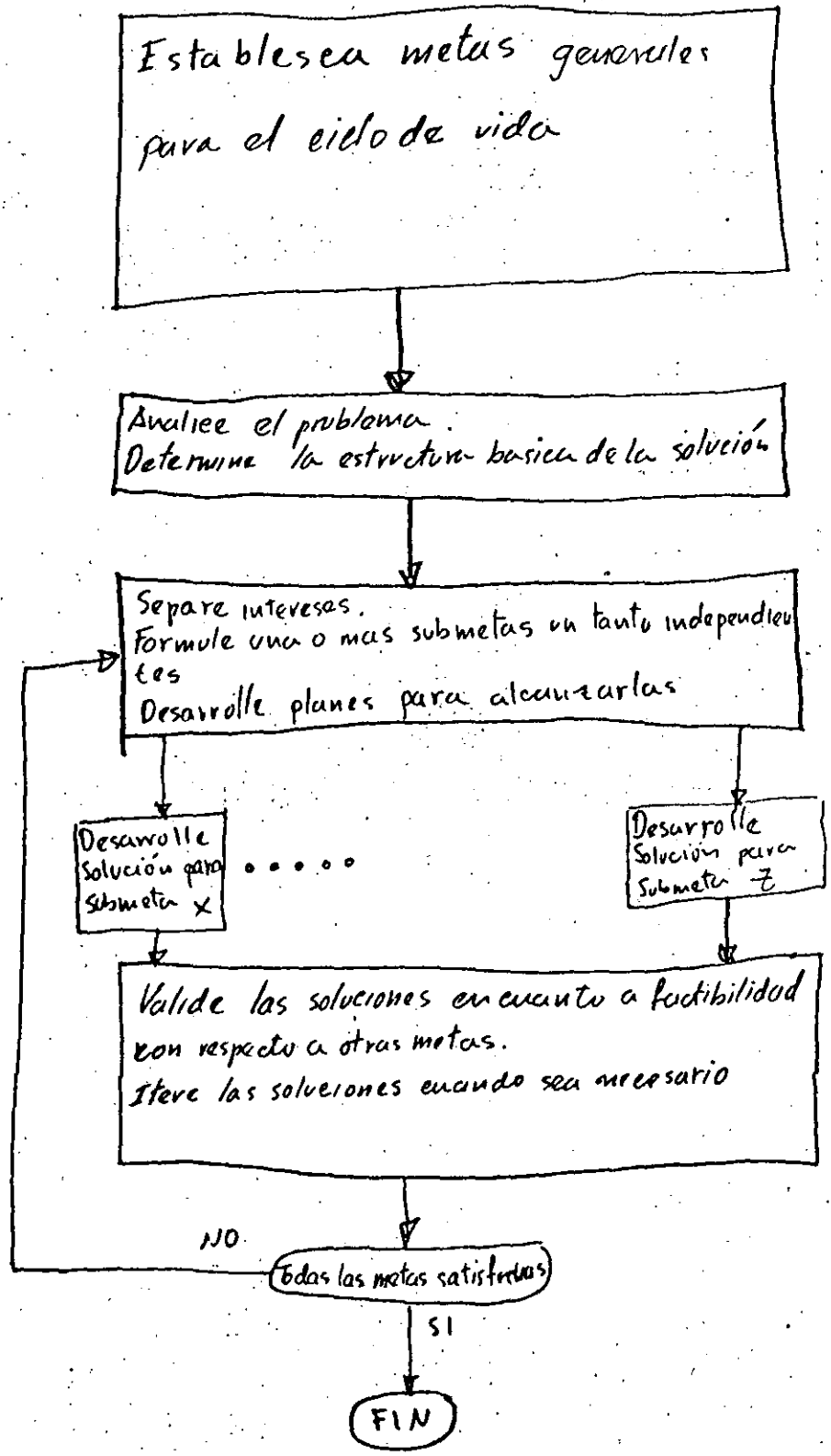
Varios niveles de Submetas pueden ser necesarios para definir los medios con suficiente detalle para mantener el proceso bajo control.

4. Siga su plan hasta el logro de la siguiente submeta del proceo

so ( o Conjunto de Submetas si estas pueden ser trabajadas en paralelo).

5. Revise el estado tanto del proceso como del producto con respecto a todas las metas y submetas definidas.
6. Itere las metas y planes tantas veces como sea necesario.
7. Continúe desempeñando los pasos 4 a 6 para submetas sucesivas del proceso hasta que el proceso este completo.
8. Independientemente de los pasos anteriores, la revisión periódica del progreso con respecto a la estructura de metas completa. Itere sus metas y planes como sea necesario.

# Enfoque orientado a metas para el ciclo de vida del Software



# Ingeniería de Software exitosa

## Producto de Software exitoso

## Proceso de Desarrollo de Software exitoso

### Relaciones Humanas

- Fácil de usar
- Satisface las necesidades Humanas
- Hace el Potencial Humano
- Sigue la Regla de Oro Modificada

### Ingeniería de Recursos

- Balanceado
- Eficientemente
- Sintonizable

### Ingeniería de Programación

- Especificado Precisamente
- Completo
- Potigudo
- Consistente
- Factible
- Probable
- Correcto
- Adaptable
- Estructurado
- Independiente de los Dispositivos
- Comprensible

### Relaciones Humanas

- Planeación
- Organización
- Staffing
- Dirección
- Control
- Automatización
- Uso de la regla de oro modificada.

### Ingeniería de Recursos

- ~~Auto~~
- Analisis de Costo Efectividad
- Planeación
- Estimación
- Control
- Encontrar Itinerarios y Presupuestos

### Ingeniería de Programación

- Factibilidad
- Validación
- Requerimientos
- Validación
- Diseño de Producto
- $V_2 V$
- Programación
- $V_3 V$
- Integración
- $V_3 V$
- Implementación
- $V_4 V$
- Mantenimiento
- $V_4 V$
- Retiro
- $V_4 V$
- Administración de la Configuración

Estructura de metas de la Ingeniería de Software

## Estructura de Metas de la Ingeniería de Software

La estructura jerárquica de metas para la Ingeniería de Software exitosa, indica que hay que prestar atención a dos subtemas principales.

1. Lograr un producto de Software exitoso.
2. Conducir un proceso exitoso tanto en el desarrollo como en el mantenimiento del Software.

Cada una de estos dos subtemas tiene tres componentes similares.

1. Relaciones Humanas. La aplicación de la ciencia y juicio humano a el desarrollo de sistemas lo cual habilite a la gente a satisfacer sus necesidades humanas y a llenar su potencial como personas.
2. Ingeniería de Recursos. La aplicación de la ciencia y matemáticas a el desarrollo de sistemas costo-efectivos.
3. Ingeniería de Programación. La aplicación de la ciencia y matemáticas a el desarrollo de programas de computadora.

El éxito de la Ingeniería de Software es el resultado de lograr un apropiado balance entre las submetas de estos componentes, tanto para el producto como para el proceso.



12

## Metas del Producto de Software

**Fácil de Usar.** Esto implica que las entradas, salidas, documentación y controles de usuario del producto sean convenientes, naturales, flexibles y dirigidos a la persona que los va a usar.

**Satisfagan las Necesidades del Humano.** Esto implica que el producto de Software debe estar bien sintonizado a las necesidades del humano en cuanto a información o instrumentos producidos por la computadora que pretende satisfacer.

**Llene el Potencial Humano.** El producto de Software provee gran reto y satisfacción en el empleo para la gente que lo usa y lo opera.

## REGLA DE ORO MODIFICADA

Haz a los otros  
lo que tu quisieras que te hicieran  
si fueras como ellos.

Metas de la Ingeniería de Recursos para el Producto de Software

Eficiente. El producto debe cubrir su propósito sin desperdicio de recursos.

Sintonizable El logro exitoso de esta submeta implica que el producto pueda ser facilmente instrumentado y medido para identificar cuellos de botella e ineficiencias y pueda ser facilmente modificado o resintonizado para tomar en cuenta cambios en la carga de trabajo los componentes de Hardware o interfases externas.

Metas de la Ingeniería de Programación para el producto de Software.

re.

Especificado precisamente. Los requerimientos funcionales, de desempeño e interfase del producto tienen que ser especificados completamente y sin ambigüedades, como un prerrequisito para el desarrollo del programa.

Características Básicas de una Especificación Precisa

Completa. Una especificación esta completa en la medida que todos sus partes estan presentes y cada parte esta completamente desarrollada.

19

**Resguardada.** En la medida que esta especifica como el Softwa  
re debe comportarse bajo todas condiciones, particular  
mente aquellas fuera de lo normal.

**Consistencia.** Una especificación es consistente en la medida -  
que no haya conflictos entre sus diversas metas y ob-  
jetivos.

**Factible.** Esto implica que los beneficios del Ciclo de Vida del  
Sistema especificado excedan los costos del Ciclo de  
Vida.

**Probable.** En la medida en la que uno pueda identificar una técni-  
ca económicamente factible para determinar cuando o no  
el Software desarrollado satisface la especificación.

**Correcto.** Esto implica que el producto de Software satisfaga exac-  
tamente las especificaciones funcionales y de interfase,  
y cubre todas las especificaciones de desempeño dentro  
de las tolerancias requeridas.

**Adaptable.** El logro de esta submeta implica que el producto de  
Software o sus componentes pueda ser fácilmente usado  
o modificado para servir a otro propósito.

15

La adaptabilidad incluye:

**Modificabilidad.** El producto facilita la incorporación de cambios.

**Portabilidad.** El producto puede ser operado fácilmente y bien en configuraciones diferentes a la actual.

**Interoperabilidad.** El producto o sus componentes pueden ser fácilmente incorporados como componentes de otros sistemas.

**Estructurado.** Un producto de Software estructurado en la medida que esta organizado de acuerdo con los siguientes principios:

1. **Abstracción.** El producto esta organizado en una jerarquía de "niveles de abstracción" cada uno de los cuales no tiene información acerca de las propiedades de otros niveles más altos, y esconden su propia información interna de los niveles superiores.

2. **Modularidad.** El producto esta organizado en módulos pequeños, coherentes e

16  
independientes.

3. Parsimonia de Componentes. El producto es  
ta construido de el menor número  
ro práctico de componentes.

Independiente de Dispositivos. El desempeño del producto no es  
afectado por el cambio de dispositivos.

Comprensible. Un producto de Software es comprensible en la me  
dida en que su propósito y operación son claros pa  
ra la persona que debe trabajar con el.

## METAS DEL PROCESO DEL SOFTWARE

### Metas para Relaciones Humanas en el Proceso del Software

Planeación. El logro exitoso de esta submeta implica el desarrollo y continuo mantenimiento del plan de proyecto del Ciclo de Vida del Software, el cual nos dice:

- . Porque se debe llevar a cabo el proyecto.
- . Que resultados debe alcanzar el proyecto.
- . Cuando los resultados deben lograrse.
- . Quien es el responsable de lograrlo.
- . Donde deberán lograrse.
- . Como deberán alcanzarse.
- . Que tanto (en recursos) tomará el alcanzarlos.
- . Mientras o asumiendo que se mantengan las siguientes condiciones.

Organización. El logro exitoso de esta submeta implica el desarrollo y continuo mantenimiento de una estructura de papeles del proyecto y responsabilidades a través del Ciclo de Vida del Software. Los principales componentes de la organización son las funciones:

18

- Delegación de Autoridad

- División del Trabajo

**Staffing.**

Esto implica la selección, reclutamiento y retención del personal apropiado para cubrir los papeles en la organización. Particularmente en esto el gerente tiene que balancear las necesidades de los diferentes Ciclos de Vida.

- El Ciclo de Vida del Producto de Software.

- El Ciclo de Vida o carrera de cada persona involucrada en el proyecto.

**Dirección.**

El logro exitoso de estas submetas involucra los siguientes objetivos:

- Motivación. La creación y mantenimiento de retos e incentivos los cuales motiven a la gente a contribuir con su mejor esfuerzo hacia el éxito del proyecto.

- Liderazgo. La comprensión por el gerente de los factores que motivan a los subordinados y la continua reflexión en este entendimiento en las acciones y decisiones tomadas por la gerencia.

**Control.** Esto involucra la medición de los logros del proyecto con respecto a el estandar de metas y planes del proyecto, y la corrección de desviaciones para asegurar el mantener los logros del proyecto de acuerdo con los planes. Planeación y Control son submetas complementarias.

**Automatización.** El uso del poder de la computadora para liberar a la gente de tareas tediosas y propensas a errores.

Metas de la Ingeniería de Recursos en el Proceso del Software.

Análisis Costo-Efectividad. Esto implica el análisis de los costos y beneficios de enfoques alternativos a el proyecto de Software, para seleccionar el más apropiado.

Planeación y Control. Estas funciones son importantes tanto en relaciones humanas como en Ingeniería de recursos. Debido a que no se puede ejercer control sin tener un buen plan y no se puede planear sin contar con estimaciones buenas de los recursos.

Metas de la Ingeniería de Programación para el Proceso del Software

Las ocho principales submetas secuenciales son:



1. Factibilidad. El lograr la definición de un concepto preferido de operación para el producto de Software, y la determinación de la factibilidad del Ciclo de Vida y su superioridad respecto a conceptos alternativos.
  
2. Requerimientos. El logro de un enunciado precisamente especificado de las funciones requeridas, interfases y desempeño del producto de Software.
  
3. Diseño del Producto. El logro de un enunciado precisamente especificado de la arquitectura general del Hardware y Software, la estructura de control, la estructura de datos del producto junto con los demás componentes necesarios como son manuales de usuarios y planes de pruebas.
  
4. Programación. El obtener un conjunto completo de componentes del Software.
  
5. Integración. El lograr un producto de Software trabajando correctamente compuesto de sus diversos componentes.
  
6. Implantación. El lograr un sistema completamente operacional

incluyendo tales objetivos como conversión de da  
tos y programas, instalación y entrenamiento.

7. Mantenimiento. Obtener una actualización del sistema de Hardwa  
re Software trabajando completamente (se repite  
para cada actualización).

8. Retiro. El lograr una transición limpia de las funciones  
desempeñadas por un producto a sus sucesores  
(si hay alguno).

9. Verificación y Validación. Una parte integral de los logros  
de cada submeta de la Ingeniería de programación  
es la verificación y validación ( V & V ) que los  
productos intermedios realmente satisfacen sus  
objetivos. Los definiremos como sigue:

Verificación: Establecer la verdad de la corres  
pondencia entre el producto de ---  
Software y su especificación.

Validación: Establecer el ajuste o valor del pro  
ducto de Software a su misión ope-  
racional.

10. Administración de la Configuración. El logro de esta submeta

implica que el producto es capaz en cualquier momento de proveer una versión definitiva del producto de Software o de cualquiera de los productos intermedios controlados. (llamados líneas base) tales como la especificación de requerimientos. Estas líneas base y las mojoneadas en Ciclo de Vida en las cuales son establecidas son fundamentales.

Estas forman una liga vital que unifica la administración y el control del proceso de Software, y la administración y el control del proceso del Software, y la administración y el control del producto de Software.

El proceso mojonera-línea base generalmente trabaja de la siguiente manera:

1. Una versión inicial intermedia o final del producto de Software es desarrollada.
2. Esta versión inicial es verificada y validada, e iterada en caso de ser necesario.

3. Una revisión formal del producto determina si el producto esta o no en forma satisfactoria para proseguir a la siguiente fase, si no se regresa a la fase 1.

4. Si el producto es satisfactorio, se le hace línea-base (esto es, se la coloca bajo un proceso de control de cambios formal).

El hacer el producto línea base tiene las tres ventajas principales siguientes:

1. No se puede hacer cambios si no hay un acuerdo de los interesados.
2. El umbral de cambios tan alto tiende a estabilizar el producto.
3. El controlador del proceso de administración de la configuración logra la meta de tener en cualquier momento una versión definitiva del producto.

La submeta de la administración de la configuración se alcanza concurrentemente con las otras submetas secuenciales de la inge-

nería de programación del Ciclo de Vida.

Estimación de los costos del software.

La estimación de costos es el eslabon entre el análisis económico y la Ingeniería de software.

Sin una idea clara del costo del desarrollo del software nos enfrentamos a los siguientes problemas:

- 1.- El personal a cargo del proyecto carece de bases para opinar acerca del presupuesto e itinerario, asi como para evaluar las diversas opciones tales como la de comprar o construir.
- 2.- El analista de software carece de fundamentos en los cuales basarse para diseñar el sistema. Lo que generalmente lleva a minimizar el costo del hardware a costa de elevar el costo del software.
- 3.- Los gerentes de proyecto carecen de bases firmes para determinar el tiempo y esfuerzo que cada fase llevará.

Exactitud del Modelado de Costos del Software.

Actualmente la capacidad para estimar los costos del software la podemos considerar razonablemente precisa.

Las razones por las que no podemos predecir los costos exactamente son entre otros:

A.- Las instrucciones fuente no son un producto uniforme, ni son la esencia del producto deseado.

B.- El software requiere la creatividad y cooperación de seres humanos cuyo comportamiento individual y en grupo es generalmente difícil de predecir.

C.- El software tiene una base más pequeña de experiencia histórica cuantitativa y es difícil agregar a la base, datos desarrollando pequeños experimentos controlados.

El Modelo de Cascada del Ciclo de Vida del Software

Las características generales del modelo de cascada en su forma actual son las siguientes:

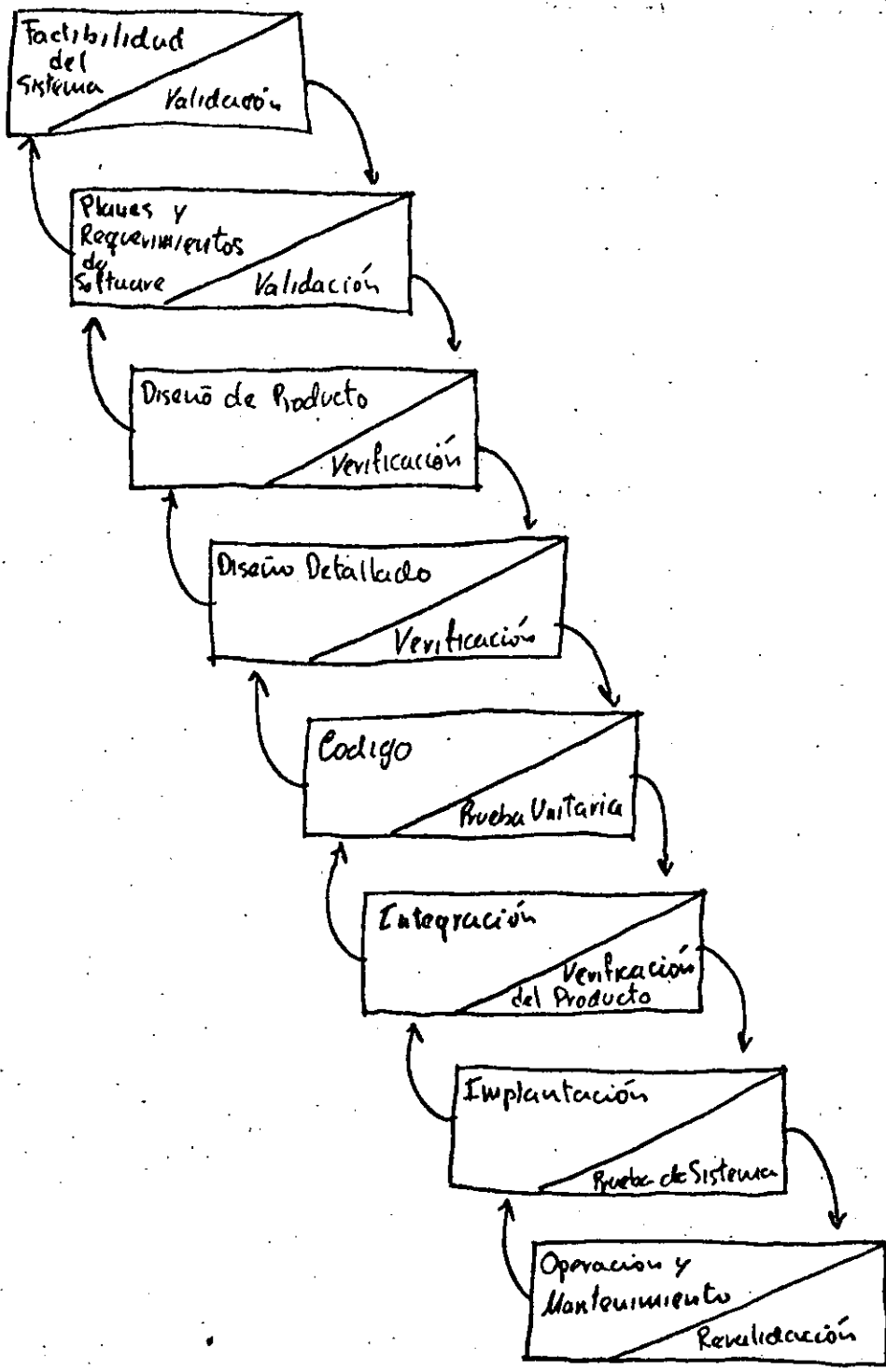
- . Cada fase es culminada por una actividad de verificación validación cuyo objetivo es eliminar - cuantos problemas sea posible en los productos de tal fase.
- . En tanto sea posible, las iteraciones de productos de fases anteriores son desempeñados en la - fase inmediata anterior.

El razonamiento Económico para el modelo de cascada se basa en dos remisas principales.

- 1.- En orden de lograr un producto de software exitoso, debemos lograr todas las submetas en alguna etapa de cualquier forma.
- 2.- Todo ordenamiento diferente de las submetas producirá un producto de software menos exitoso.



# Modelo de Cascada de el Desarrollo



## Refinamientos al Modelo de cascada

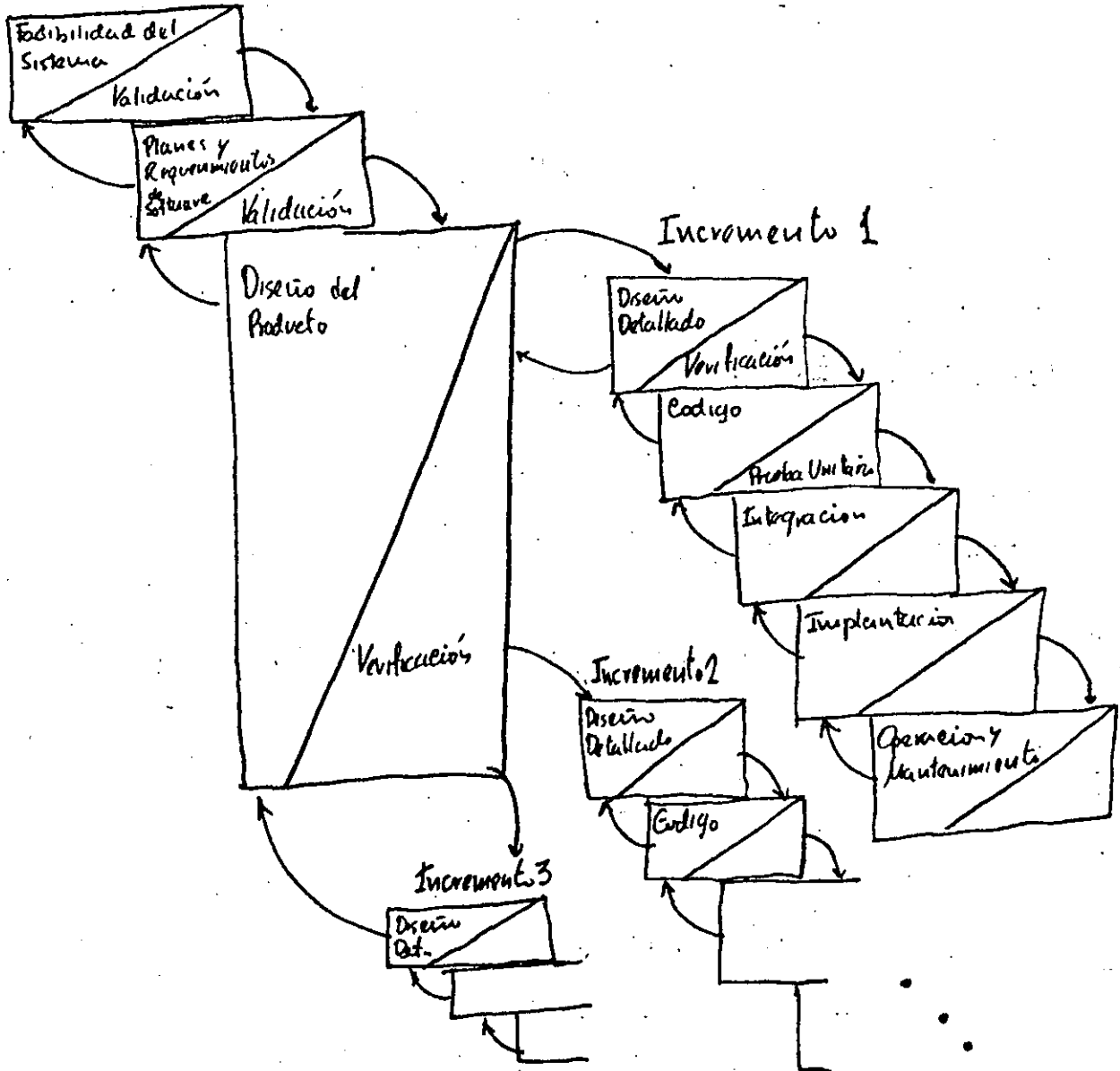
### Desarrollo Incremental

El desarrollo incremental es un refinamiento de tanto el enfoque de prototipo completo o hagalo dos veces y enfoque - top-down, nivel por nivel. Este sostiene que en vez de los dos enfoques anteriores, debemos desarrollar el software en incrementos de capacidad funcional

La principal ventaja del desarrollo incremental sobre los otros enfoques son los siguientes:

- Los incrementos de capacidad funcional son mucho más útiles y fáciles de probar que los productos de nivel intermedio en el desarrollo top-down nivel - por nivel.
- El uso de los incrementos sucesivos provee un camino para incorporar la experiencia del usuario en un producto refinado en una forma menos costosa que el completo desarrollo duplicado en el enfoque hagalo dos veces.

# Modelo de Cascada usando Desarrollo Incremental



## Documentación avanzada

Documentación preparada en avance por dos razones

- 1.- Para definir objetivos detallados y planes para actividades futuras de desarrollo de software.
- 2.- Para producir versiones tempranas de documentación de usuario. Esto tiene una gran ventaja ya que da a los usuarios la oportunidad de ver como el sistema los afectará.

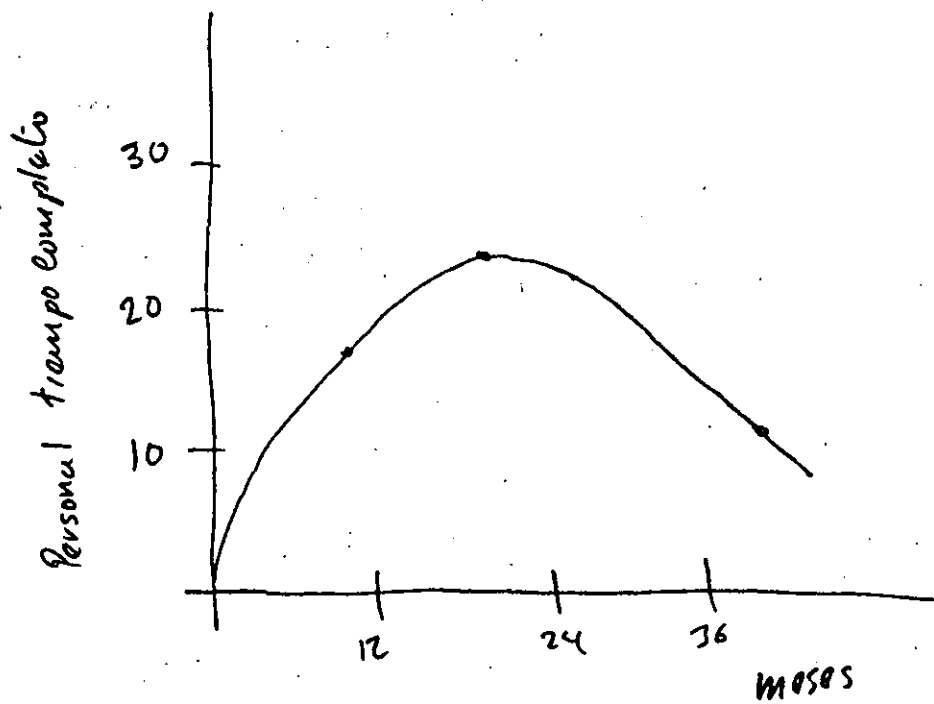
## Andamiaje

Andamiaje se refiere a los productos extras que deberán ser desarrollados para realizar la tarea principal de desarrollo de software y V&V avanzar suavemente y tan eficientemente como sea posible.

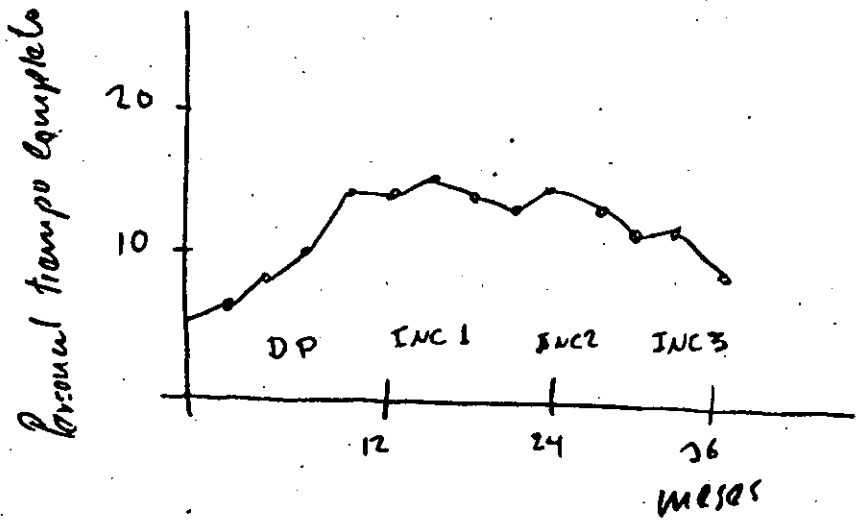
## Implicaciones Económicas

- 1.- Tienden a reducir los costos generales, primeramente reduciendo la entropía involucrada en el ciclo de vida del software: aquellas actividades las cuales consumen la energía de la gente y talento sin resultados constructivos.
- 2.- Tienden a adelantar la distribución de la carga de trabajo.

# Distribución de la Fuerza de Trabajo a lo largo del proyecto



curva de Rayleigh



desarrollo incremental

## ACTIVIDADES Y FASES

Para propósitos de planeación presupuestaria y control es útil organizar las actividades del proyecto en unas estructuras jerárquicas llamadas estructuras del despiece del trabajo.

Jerarquía del producto, la cual indica como los varios componentes del software ajustan en el sistema de software completo.

Jerarquía de actividades, la cual indica las varias actividades, las cuales pueden tratar con un componente de software.

En la práctica tanto en la jerarquía del producto como en la jerarquía de actividades asociadas para cada componente, son definidas únicamente al nivel necesario para reporte de costos y control, en términos del tamaño del proyecto podemos decir

Pequeño	7 hombres-mes	al menos 7%	ó	0.5	hombres-mes
Mediano	300 hombres-mes	al menos 1%	ó	3	hombres-mes
Muy grande	7000 hombres-mes	al menos 0.2%	ó	15	hombres-mes

man-month project to develop an energy model of about 10,000 instructions. It shows some activities applied at more than one level of the product hierarchy. Management is applied both to the overall project (S1) and to the computation subsystem (SB1). System engineering is applied both to the overall project (S2, with components S21 and S22) and to the energy module (SBA2).

### Uses of the Software WBS

One main use of the software WBS is to help define just what costs are being estimated by a software cost-estimation model. Without such definitions, software cost estimates and data lose precision and meaning. The dotted lines in Fig. 4-6b show that the software development costs estimated by the COCOMO model presented in this book cover all of the work performed in the first five major activity elements (SX1-SX5), with the exception of feasibility studies—the work performed in the feasibility phase of the software life-cycle—and requirements analysis, which is estimated as a separate quantity apart from software development.

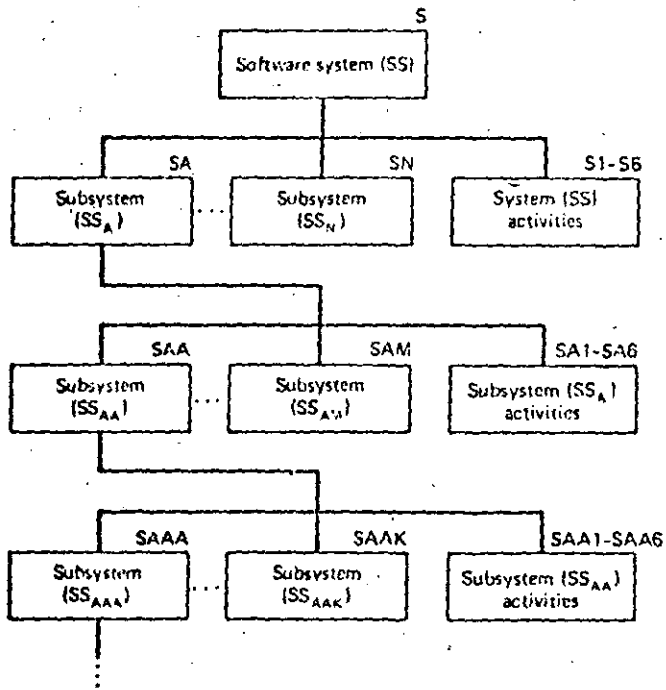


FIGURE 4-6(a) Software work breakdown structure: Product hierarchy

DEFINICION DE FIN DE LAS FASES

- 1 Inicio de planes y Fase de requerimientos  
 (Fin de la revisión de conceptos del ciclo de vida)
  - Aprobación de la arquitectura del sistema validada, incluyendo asignaciones básicas de hardware-software.
  - Aprobación de la validación del concepto de operación incluyendo asignaciones básicas hombre-máquina.
  - Plan del ciclo de vida de alto nivel, incluyendo mojoneras, recursos, responsabilidades, itinerarios y actividades principales.
  
- 2 Fin de Fase de Planes y Requerimientos. Inicio de la fase de Diseño del Producto.  
 (Fin de la revisión de requerimientos de software)
  - Plan de desarrollo detallado; criterios detallados de mojoneras de desarrollo, presupuesto de recursos, organización, responsabilidades, itinerarios, actividades, técnicas y productos.
  - Plan detallado de uso; contrapartes de los puntos del plan de desarrollo para entrenamiento, conversión, instalación, operaciones y reporte.



- Plan detallado de Control del Producto; plan de administración de la configuración, plan de aseguramiento de la calidad, plan general de V&V (excluyendo planes detallados de pruebas)
  
- Aprobación de las especificaciones de requerimientos de software validadas; funcionales, desempeño, y especificaciones de interfase validadas por completas, consistentes, probables y factibles.
  
- Aprobación (formal o informal) del contrato de desarrollo; basado en los puntos anteriores.

3 Fin de la Fase de Diseño del Producto. Inicio de la Fase de diseño detallado.

(Fin de la Revisión del diseño del producto)

- Especificación del diseño del producto especificado
  - Jerarquía de componentes del programa, interfase de control y datos hasta nivel unitario.
  - Estructura de datos lógica y física hasta el nivel de campo.
  - Presupuesto de recursos para procesamiento de datos
  - Verificado por completo, consistente, factible y rastreable en los requerimientos.

- Identificación y resolución de los puntos de desarrollo de alto riesgo.

- Integración preliminar y plan de pruebas  
Plan de pruebas de aceptación y manuales de usuario.

4 Fin de la Fase de Diseño Detallado: Inicio de la codificación y Fase de pruebas unitarias.

(Fin de los walkthrough de diseño o revisión de Diseño por unidad))

- Especificación de diseño detallado verificada por cada unidad.

. Por cada rutina ( 100 instrucciones fuente), especifique nombre, propósito, supuestos, tamaño, secuencia de llamado, salidas de error, entradas, salidas, algoritmos y flujo de procesamiento.

. Descripción de la Base de Datos a nivel de parámetro/carácter/bit.

. Verificado por completo, consistente y rastreable con requerimientos y especificaciones de diseño del sistema y presupuestos.

- Plan de pruebas de aceptación aprobado.

- Bosquejo completo del plan de integración y pruebas y manuales de usuario.

5 Fin de la Fase de Codificación y Prueba unitarias. Inicio de la Fase de pruebas e Integración.

(Satisfacción de los criterios de pruebas unitarias)

- Verificación de todas las unidades computacionales, usando no únicamente valores nominales sino también valores singulares y extremos.
- Verificación de todas las opciones de entrada y salida de todas las unidades, incluyendo mensajes de error.
- Ejercitación de todos los enunciados ejecutables y todas las opciones de ramificación.
- Verificación de la adherencia a los estándares de programación.
- Finalización de la documentación a nivel unitario.

6 Fin de la Fase de Integración y pruebas, Inicio de la Fase de Implantación

(Fin de la revisión de aceptación del software)

- Satisfacción de la prueba de aceptación del software.

- . Verificación o satisfacción de los requerimientos de software.
- . Demostración del desempeño.

- Aceptación de todos los productos entregables: reportes, manuales, especificaciones como se construyo, Bases de Datos.

7 Fin de la Fase de Implantación. Inicio de la Fase de operación y mantenimiento.

(Fin de la Revisión de aceptación del sistema)

- Satisfacción de las pruebas de aceptación del sistema
  - . Verificación de la satisfacción de los requerimientos del sistema.
  - . Verificación del estado operacional del software, hardware, personal e instalaciones.

- Aceptación de todos los productos del sistema entregables: hardware, software, documentación, entrenamiento e instalaciones.

- Finalización de todas las actividades de conversión e instalaciones especificadas.

8 Fin de la Fase de Operación y Mantenimiento

(Vía retiro)

- Finalización de todos los puntos en el plan de retiro: conversión, documentación, archivo, transición al nuevo sistema.

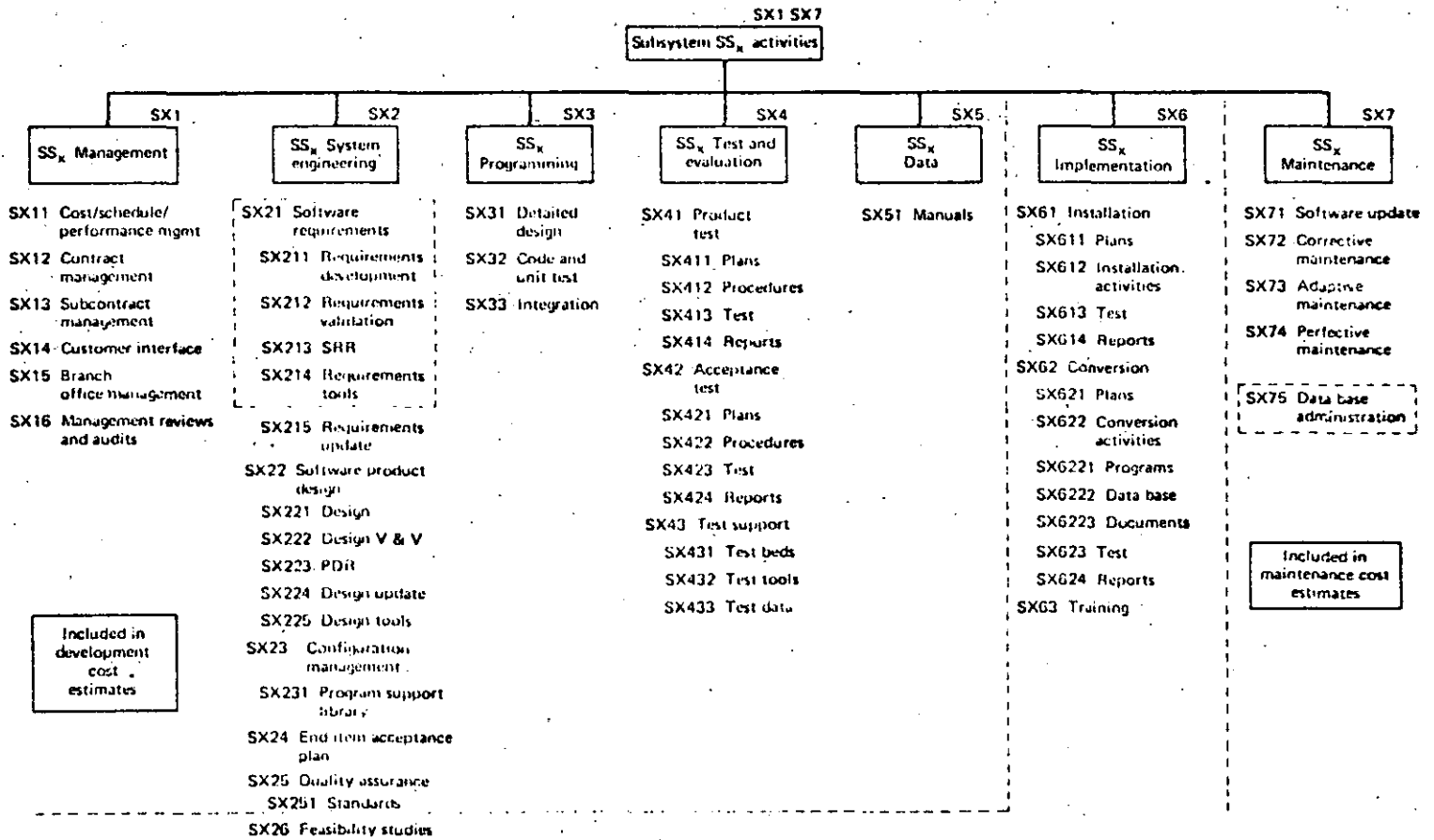


FIGURE 4-6(b) Software work breakdown structure (WBS): Activity hierarchy

Definición de Actividades

Análisis de Requerimientos	Determinación, especificación, revisión y actualización del software funcional. requerimientos de desempeño, interfase y verificación.
-------------------------------	--

Diseño del Producto	Determinación, especificación, revisión y actualización de la arquitectura de hardware-software, diseño de programas y diseño de la Base de Datos.
------------------------	--

Programación	Diseño detallado, codificación, prueba unitaria e integración de los componentes individuales de los programas. Incluye - planeación del personal de programación, adquisición de herramientas, desarrollo - de la Base de Datos, documentación a nivel de componente y administración de la programación a nivel medio.
--------------	--

Planeación de Pruebas	Especificación, revisión, y actualización de los planes de pruebas de aceptación y pruebas de producto. Adquisición de las herramientas y datos de prueba.
--------------------------	--

Verificación  
y  
Validación

Validación de requerimientos de desempeño independientes, V&V de diseño, pruebas de producto y pruebas de aceptación. Adquisición de herramientas.

Funciones de la  
Oficina de Pro-  
yectos

Funciones de administración a nivel de proyecto. Incluyen planeación y control a nivel de proyecto, contratación y subcontratación de administración e interfase con el cliente.

Administración  
de la Configu-  
ración y asegu-  
ramiento de la  
calidad

La administración de la calidad incluye identificación del producto, control de cambios, contabilidad de status, operación de la librería de soporte a la programación, desarrollo y monitoreo del plan de aceptación. El aseguramiento de la calidad incluye el desarrollo y monitoreo de estándares de proyecto y auditorías técnicas a los productos y procesos del software.

Manuales

Desarrollo y actualización de los manuales de usuario, manuales de operador y manuales de mantenimiento.



**TABLE 4-3 Project Tasks by Activity and Phase**

Activity	Phase			
	Plans and Requirements	Product Design	Programming	Integration and Test
Requirements analysis	Analyze existing system, determine user needs, integrate, document, and iterate requirements	Update requirements	Update requirements	Update requirements
Product design	Develop basic architecture; models, prototypes, risk analysis	Develop product design; models, prototypes, risk analysis	Update design	Update design
Programming	Top-level personnel and tools planning	Personnel planning, acquire tools, utilities	Detailed design, code and unit test, component documentation, integration planning	Integrate software, update components
Test planning	Acceptance test requirements, top-level test plans	Draft test plans, acquire test tools	Detailed test plans, acquire test tools	Detailed test plans, install test tools
Verification and validation	Validate requirements, acquire requirements, design V & V tools	V & V product design, acquire design V & V tools	V & V top portions of code, V & V design changes	Perform product test, acceptance test, V & V design changes
Project office functions	Project level management, project MIS planning, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.
CM/OA	CM/OA plans, procedures, acceptance plan, identify CM/OA tools	CM/OA of requirements, design; project standards, acquire CM/OA tools	CM/OA of requirements, design; code, operate library	CM/OA of requirements, design; code, operate library; monitor acceptance plan
Manuals	Outline portions of users' manual	Draft users', operators' manuals, outline maintenance manual	Full draft users' and operators' manuals	Final users', operators', and maintenance manuals

Figure 5-1 also summarizes the underlying software development process model which COCOMO assumes will be used on the project. This process emphasizes the following major features:

1. Careful definition and validation of the software requirements specification by a relatively small number of people prior to significant work on the full system design.

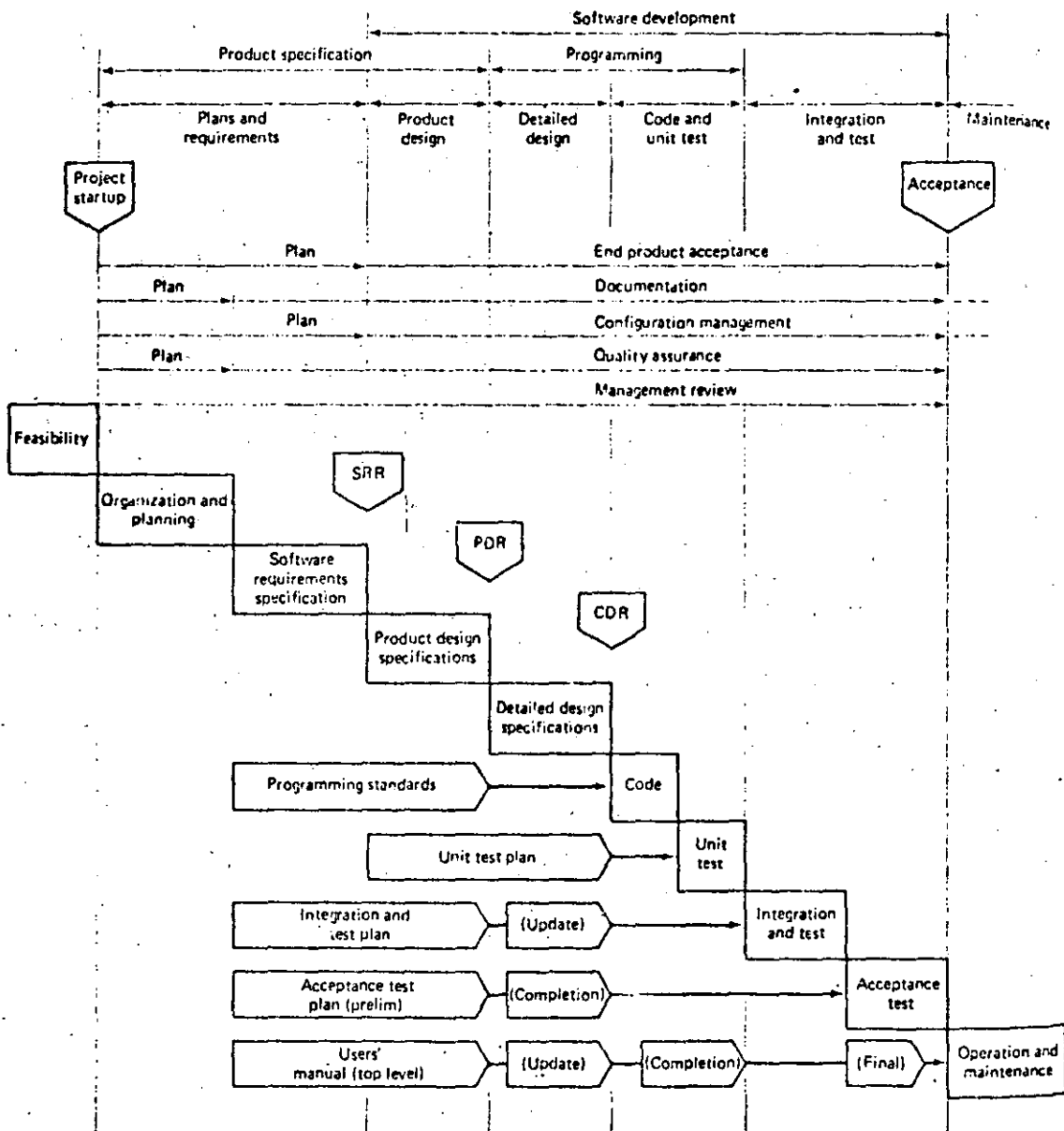


FIGURE 5-1 Software project phases, activities, and milestones

2. C  
u  
f  
3. I  
r  
c  
4. J  
e  
5.  
Ma  
CC  
between  
costs?  
months  
intern  
In  
best cc  
dollar  
differer  
average  
design  
for the  
5.3  
We ha  
to giv  
This  
the m  
oped  
form  
matin  
The  
Chap

FIGURE 4-6(b) Software work breakdown structure (WBS): Activity hierarchy

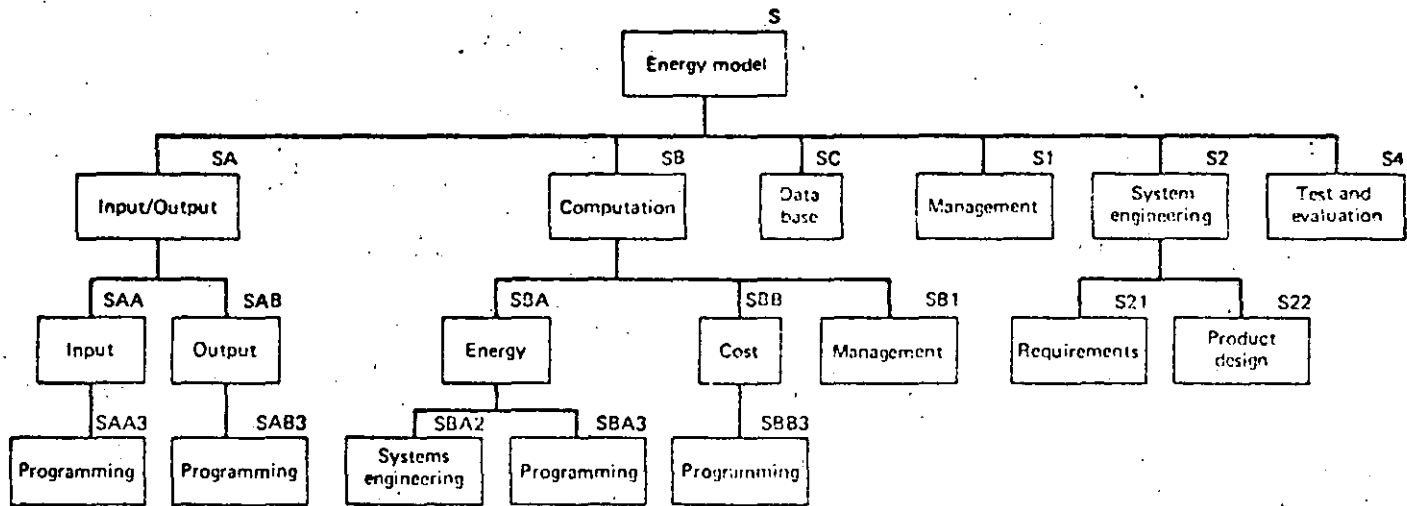


FIGURE 4-7 Example software work breakdown structure

## Mantenimiento de Software

El mantenimiento de Software se define como el proceso de modificar el Software Operacional existente dejando sus funciones primarias intactas. La definición incluye las siguientes tipos de actividades dentro de la categoría de mantenimiento de Software.

- Rediseño y rediseño de pequeñas porciones (menos de 50% de código nuevo de un producto de Software existente.
- Diseño y desarrollo de pequeños paquetes de interfase de -- Software los cuales requiera algo de rediseño (de menos del 20%) del producto de Software existente.
- Modificación del código del producto de Software su documentación o estructura de la Base de Datos.

El mantenimiento de Software puede ser clasificado en dos categorias principales:

1. Actualización de Software, el cual resulta en un cambio en la especificación funcional del producto.
2. Reparación de Software, la cual deja la especificación funcional intacta.

Las Reparaciones puede clasificarse en 3 subcategorías:

- 2a. Mantenimiento correctivo (de procesamiento, o fallas de implantación).
- 2b. Mantenimiento adaptivo ( a cambios en el procesamiento o medio ambiente de datos).
- 2c. Mantenimiento perfectivo (para mejorar el desempeño o la posibilidad de mantenimiento)

Modelo Básico

Definiciones y Supuestos

1. Instrucciones Fuente Entregadas

(Delivered source instructions (DSI) )

Entregadas. Este termino generalmente significa que se excluye Software que no es entregado. Sin embargo si es desarrollado con el mismo cuidado que el Software entregado, con sus revisiones, planes de prueba, documentación, etc., ellas deberán contarse.

Instrucciones Fuente. Este termino incluye todas aquellas ins-trucciones de programa creadas por el personal de proyecto y procesados en la máquina por alguna combinación de preprocesadores, compiladores y ensam-bladores.

Esto excluye comentarios y tarjetas de utileria sin modificaciones. Incluye tarjetas de JCL, enunciados de formato y declaraciones de datos. Instrucciones se definen como líneas de código o imagenes de tar-jeta.

- 2. El período de desarrollo cubierto por el modelo inicia en la fase de diseño del producto, y termina con la fase de pruebas de integración.
- 3. Las estimaciones de costo cubren solo aquellas actividades indicadas en la figura.
- 4. El modelo de costos solamente cubre los costos directos de salarios.
- 5. En el modelo un hombre mes consiste de 152 horas de tiempo trabajado.

$$\text{Hombres} - \text{Hora} = \text{Hombres} - \text{Mes} \times 152$$

$$\text{Hombres} - \text{Día} = \text{Hombres} - \text{Mes} \times 19$$

$$\text{Hombres} - \text{Año} = \text{Hombres} - \text{Mes} - 12$$

- 6. El modelo asume una buena administración tanto por parte del que desarrolla como del cliente.
- 7. El modelo asume especificaciones de requerimientos relativamente estables.
- 8. Los Modelos Básicos e Intermedio no distinguen entre los factores de costo de las fases únicamente entre desarrollo y mantenimiento.
- 9. Los costos de las fases incluyen todos los costos en que se incurre durante la fase.

La siguiente figura también resume el modelo fundamental del proceso de desarrollo de Software que el modelo de costo asume, se-  
rá usado en el proyecto.

Este proceso enfatiza las siguientes características principales:

1. La cuidadosa definición y validación de la especificación de requerimientos de Software por un relativamente pequeño número de gentes antes de trabajo en el diseño del sistema completo.
2. La cuidadosa definición y validación del diseño del Software del sistema hasta el nivel unitario por un grupo mayor, pero aún relativamente pequeño, antes de realizar un trabajo significante en el diseño detallado y codificación.
3. Diseño detallado, codificación y pruebas unitarias desempeñado por un grupo de programadores en paralelo, trabajando dentro de un marco de diseño del sistema con líneas-base firmes frecuentemente con respecto a un desarrollo incremental planeado.
4. La Integración y prueba de cada incremento es basado en un monto significativo de planeación de pruebas temprana, y la eliminación de casi todas las fallas dentro de las unidades por medio de walk throughs y pruebas unitarias.
5. Mucho del esfuerzo de la documentación es desempeñado de



forma temprana, en orden de proveer a los usuarios (y a los que desarrollan) alguna realimentación temprana en la naturaleza operacional del producto.

Esfuerzo de Desarrollo & Schedule

Esfuerzo de Desarrollo

$$MM = 2.4 (KDSI)^{1.05}$$

MM=hombres-mes

KDSI= miles de instrucciones fuente entregadas

Schedule

$$TDEV = 2.5 (MM)^{0.38}$$

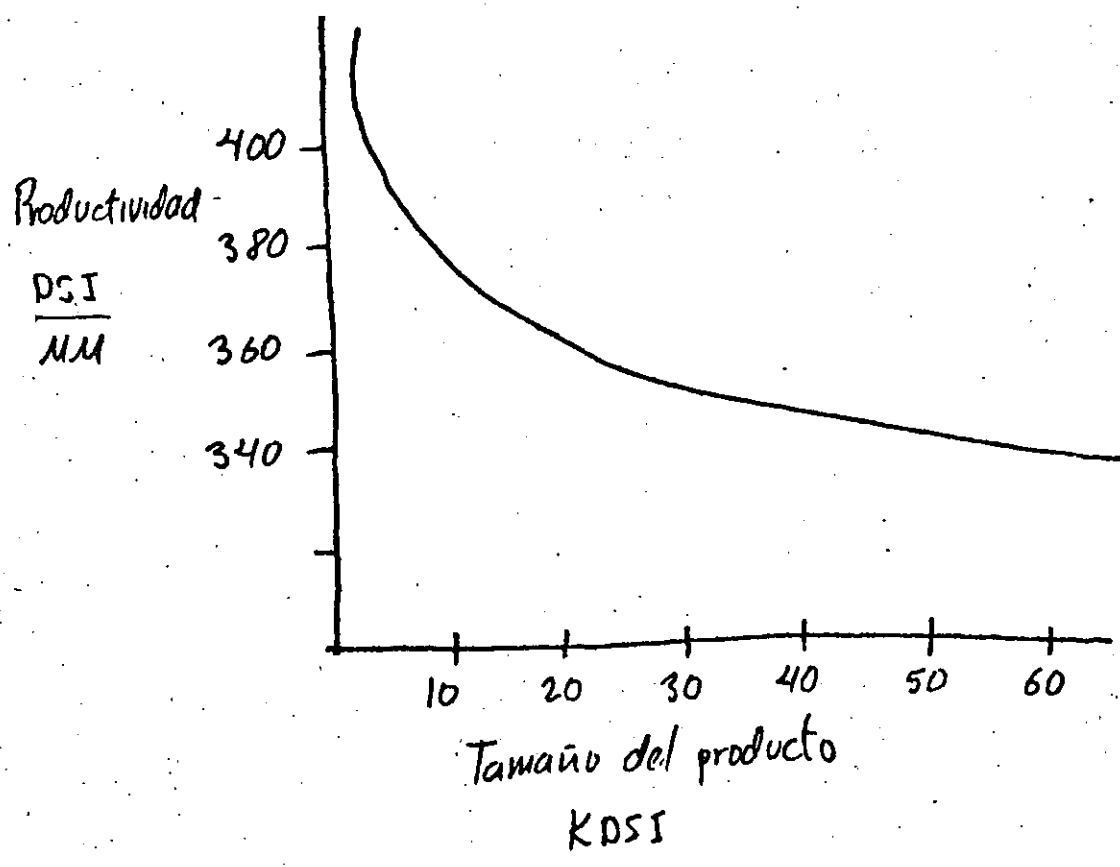
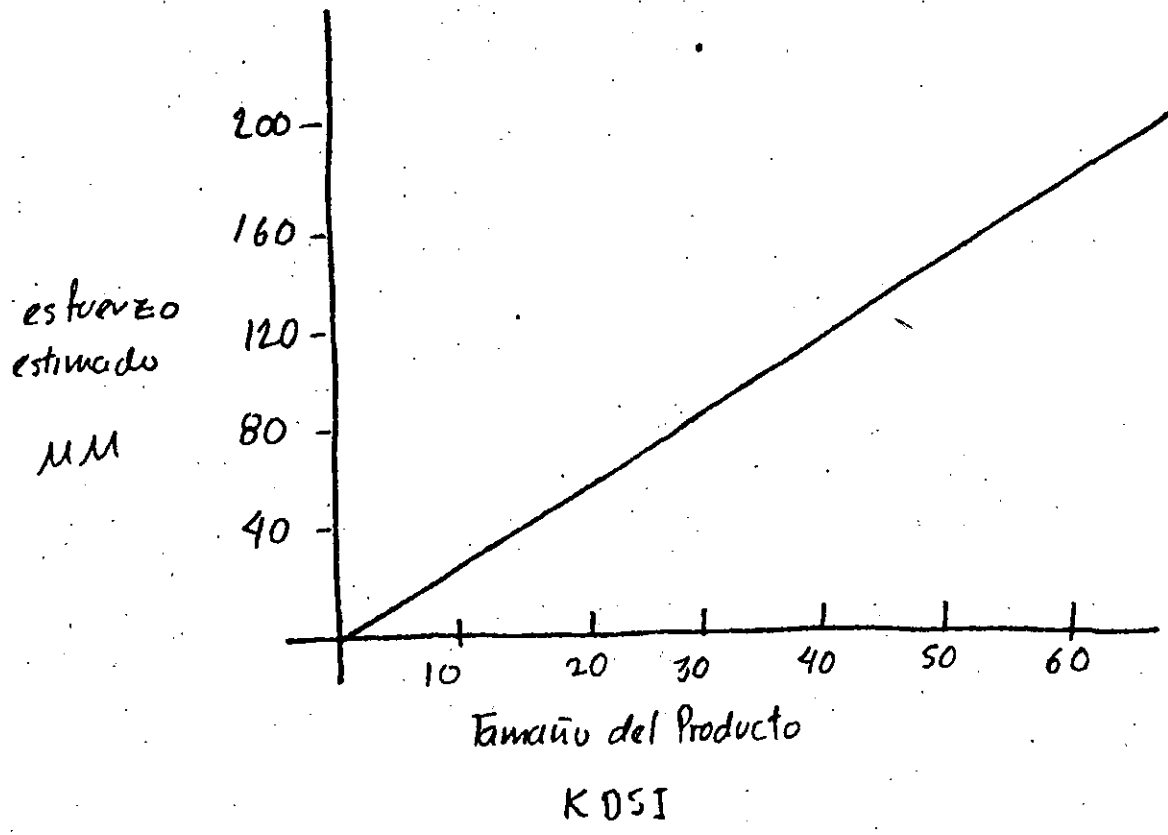
TDEV=tiempo de desarrollo en meses.

Productividad

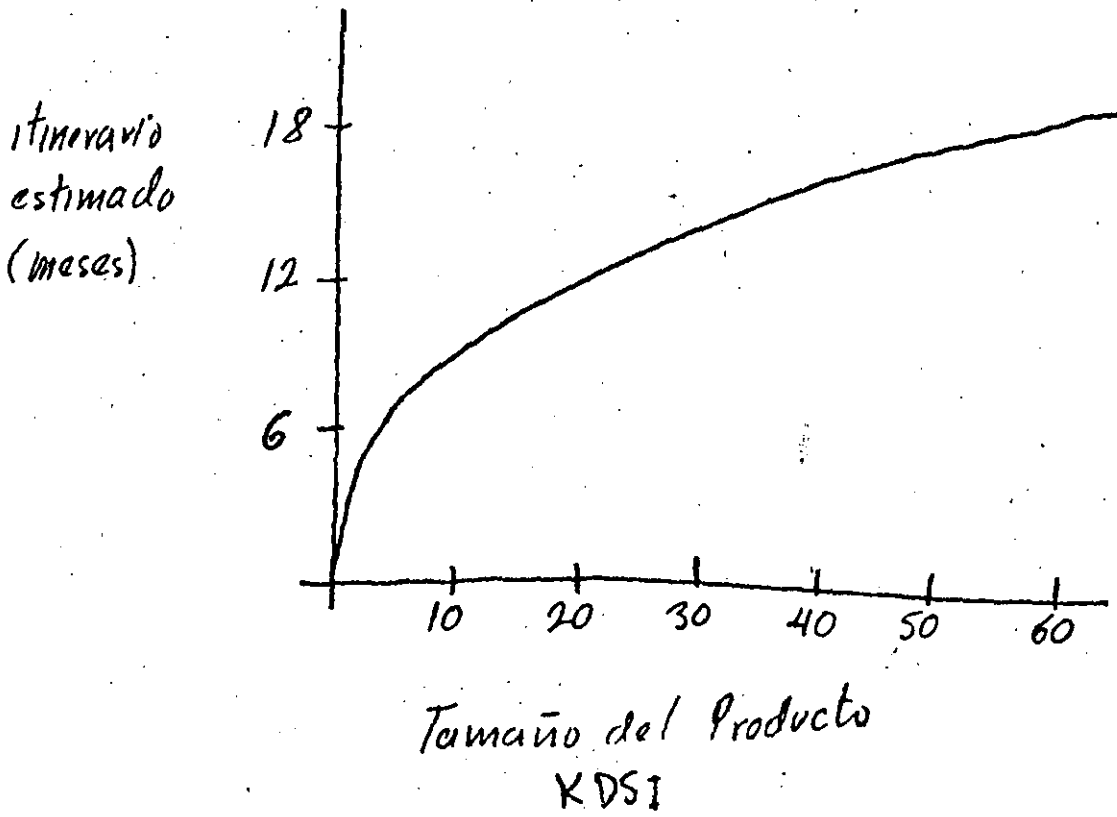
$$\frac{DSI}{MM}$$

Personal promedio

$$\frac{MM}{TDEV}$$



Resumen del Modelo: Modo Organico



### Perfiles de Proyecto

Modo Organico

Tamaño del producto	Esfuerzo	Productividad	Itinerario (meses)	Personal Promedio	
Pequeño	2 KDSI	5.0 MM	400 DS/MM	4.6	1.1
Intermedio	8 KDSI	21.3 MM	376 DS/MM	8.0	2.7
Mediano	32 KDSI	91.0 MM	352 DS/MM	14.0	6.5
Grande	128 KDSI	392.0 MM	327 DS/MM	24.0	16.0

Distribución por fase de esfuerzo y Itinerario : Modo Organico

Fase	tamaño del producto			
	pequeño (2KDSI)	Intermedio (8KDSI)	Mediano (32KDSI)	Grande (128KDSI)
<b>Esfuerzo</b>				
Planes y requerimientos	6%	6%	6%	6%
Diseño del producto	16	16	16	16
Programación	68	65	62	59
Diseño detallado	26	25	24	23
Codificación y prueba unitaria	42	40	38	36
Integración y prueba	16	19	22	25
<b>Total</b>	<b>100%</b>	<b>100%</b>	<b>100%</b>	<b>100%</b>
<b>Itinerario</b>				
Planes y requerimientos	10%	11%	12%	13%
Diseño del producto	19	19	19	19
Programación	63	59	55	51
Integración y pruebas	18	22	26	30
<b>Total</b>	<b>100%</b>	<b>100%</b>	<b>100%</b>	<b>100%</b>

# Perfiles Básicos del Proyecto: Modo Organico

## Tamaño del Producto

Cantidad Pequeño (2KDSI) Intermedio (8KDSI) Mediano (12KDSI) Grande

Esfuerzo total (um)	5.0	21.3	91	392
Planes y Requerimientos	0.3	1.3	5	24
Diseño del Producto	0.8	3.4	15	63
Programación	3.4	13.8	56	231
Diseño Detallado	1.3	5.3	22	90
Codificación y prueba unitaria	2.1	8.5	34	141
Integración y pruebas	0.8	4.1	20	98
Itinerario Total (meses)	4.6	8	14	24
Planes y Requerimientos	0.5	0.9	1.7	3.1
Diseño del producto	0.9	1.5	2.7	4.6
Programación	2.9	4.7	7.7	12.2
Integración y Pruebas	0.8	1.8	3.6	7.2
Personal Promedio				
Planes y Requerimientos	0.6	1.4	2.9	8
Diseño del Producto	0.9	2.3	5.6	14
Programación	1.2	2.9	7.3	19
Integración y Pruebas	1.0	2.3	5.6	14
Promedio del proyecto	1.1	2.7	6.5	16
Porcentaje del promedio del proyecto				
Planes y Requerimientos	60%	55%	50%	46%
Diseño del Producto	84	84	84	84
Programación	108	110	113	116
Integración y Pruebas	89	87	85	83
Productividad (DSI/um)	400	376	352	327

Al decremento en la productividad en proyecto grandes se le llama deseconomía de escala.

Las principales razones por las que productos grandes de Software incurren en deseconomías de escala son:

1. Relativamente más diseño de producto es requerido para desarrollar hasta el nivel de unidad las especificaciones requeridas para soportar la actividad paralela de un gran grupo de programadores.
2. Se requiere relativamente más esfuerzo para verificar y validar el número mayor de requerimientos y especificaciones de diseño.
3. Aún y cuando se cuenta con especificaciones más completas los programadores de proyectos grandes ocuparan relativamente más tiempo comunicando y resolviendo asuntos de interfase.
4. Relativamente más actividad de integración se requiere para poner las unidades juntas.
5. En general, se requieren pruebas más extensivas para verificar y validar el producto.
6. Se requiere relativamente más esfuerzo para administrar el proyecto.

Estimación Básica del Esfuerzo de Mantenimiento del Software

Tráfico anual de cambios (ACT)

(Annual Change Traffic)

La fracción de instrucciones fuente del producto las cuales sufren cambios durante un año (típico), ya sea por adición o modificación.

$$(MM)_{AM} = 1.0 (ACT)(MM)_D$$

$(MM)_{AM}$  = hombres-mes ocupados en mantenimiento anual

$(MM)_D$  = hombres-mes de esfuerzo de desarrollo estimado.

Modelo Básico: Modos de Desarrollo

- Existen varios modos de desarrollo de Software.
- Estos diferentes modos de desarrollo de Software tienen relaciones de estimación de costos que son de forma similar, pero que dan estimaciones de costos significativamente diferentes para productos de Software del mismo tamaño.

Modo Orgánico

- Equipos de desarrollo relativamente pequeños desarrollan el Software en un ambiente altamente familiar, desarrollo en cara.
- La mayoría del personal conectado con el proyecto tiene experiencia extensiva trabajando con sistemas relacionados dentro de la organización, y tienen una clara idea de como el sistema en desarrollo contribuirá a los objetivos de la organización.
- Un medio ambiente de desarrollo generalmente estable, con un desarrollo concurrente de Hardware nuevo y procedimientos operacionales asociados muy pequeño.
- Necesidad mínima de algoritmos y arquitecturas de procesamiento de datos nuevos.



60

Un premio relativamente bajo para adelantar la fecha en que se termine el proyecto.

Tamaño relativamente pequeño. Muy pocos son los proyectos de modo orgánico que han desarrollado productos con más de 50 KDSI de Software nuevo.

Modo Semiseparado

El modo semiseparado de desarrollo de Software representa una eta para intermedia entre los modos orgánico y el incrustado (embedded).

Intermedio puede significar cualquiera de dos cosas:

- 1. Un nivel intermedio de las características del proyecto
- 2. Una mezcla de las características de los modos orgánicos e incrustados.

Así con respecto a la característica "experiencia en trabajo con sistemas de Software relacionados", cualquiera de las siguientes puede ser una característica de un proyecto de modo semiseparado.

- . Los miembros del equipo tienen todos un nivel de experiencia medio con sistemas relacionados.
- . El equipo tiene una amplia mezcla de personas con experiencia y sin ella.
- . Los miembros del equipo tienen experiencia relacionada con algunos aspectos del sistema bajo desarrollo, pero no otros.

Los proyectos de modo-semiseparado pueden ser un sistema de procesamiento de transacciones con algunas interfases muy rigurosas y otros no tanto.

62

La flexibilidad parcial explica el termino semiseparado.

El Rango de tamaño de productos de modo semiseparado generalmente se extiende hasta los 300 KDSI.

### Modo Incrustado.

El factor que distingue este tipo de Software es la necesidad de operar dentro de restricciones muy estrechas.

El producto debe operar dentro (esta incrustado en) de complejos fuertemente acoplados de Hardware, Software, Regulaciones y Procedimientos Operacionales.

En general, los costos de cambiar las otras partes de estos complejos son tan altos que sus características son consideradas esencialmente incambiables, se espera que el Software se conforme a sus especificaciones y absorba cualquier dificultad no prevista o cambios requeridos dentro de las otras partes del complejo.

- . Este modo requiere de un pequeño grupo de analistas en las etapas tempranas.
- . Una vez se ha terminado el diseño del producto traer a un gran equipo de programadores a realizar el diseño detallado, codificación y pruebas unitarias en paralelo. De otro modo el -- proyecto llevaría más tiempo, lo que sería negativo por las si guientes razones:

- 107
- El producto tendría que absorber más cambios.
  - El producto se entregaría fuera de tiempo.

Estas estrategias ocasionan picos en la demanda de personal en el proyecto.

## Funciones Basicas de Esfuerzo e Itinerario

Modo	Esfuerzo	Itinerario
Organico	$MM = 2.4(KDSE)^{1.05}$	$TDEV = 2.5(MM)^{0.38}$
Semiseparado	$MM = 3.0(KDSE)^{1.12}$	$TDEV = 2.5(MM)^{0.35}$
Incrustado	$MM = 3.6(KDSE)^{1.20}$	$TDEV = 2.5(MM)^{0.32}$

TABLE 6-3 Distinguishing Features of Software Development Modes

Feature	Mode		
	Organic	Semidetached	Embedded
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300 KDSI	All sizes
Examples	Batch data reduction Scientific models Business models Familiar OS, compiler Simple inventory, production control	Most transaction processing systems New OS, DBMS Ambitious inventory, production control Simple command-control	Large, complex transaction processing systems Ambitious, very large OS Avionics Ambitious command-control

*Semidetached Mode* (Aircraft flight-training simulator): "We need a fair amount of accuracy for this flight simulator, and the sensor inputs are somewhat different from our previous experience, but our main concern will be in getting the simulated aircraft position computed in time for each display cycle. If we have to reduce accuracy to meet the time constraints, we can live with that."

*Embedded Mode* (Aircraft on-board collision avoidance system): "With this new radar, we're going to have to experiment with various algorithms to find one with adequate accuracy and speed. If we don't find a satisfactory algorithm, we'll have to rework the entire collision-avoidance approach in the on-board computer, and we may run into problems with the FAA flight-safety guidelines."

A large software project may contain several subprojects operating in different modes (embedded-mode mission control software; organic-mode support software.) Further, if the subprojects are not closely interrelated and do not cause each other

TABLE 6-4 Project Activity Differences Due to Software Development Mode

Mode	Phase			
	Requirements and Product Design	Detailed Design	Code and Unit Test	Integration and Test
Embedded	Extensive rework to accommodate specification changes Thorough specification, validation of requirements and interfaces Extensive analysis, prototyping of high-risk elements	Very formal configuration management, interface control	Extensive rework to accommodate code changes	Extensive requirements, interface testing
Semidetached	Intermediate level of above effects			
Organic	Relatively little rework to accommodate specification changes Fairly general specification, validation of requirements and interfaces Occasional analysis, prototyping of high-risk elements	Fairly informal configuration management, interface control	Moderate rework to accommodate code changes	Moderate requirements, interface testing



diseconomies of scale, their costs should be estimated as several smaller projects rather than as one large project with its corresponding large diseconomy of scale.

## 6.4 DISCUSSION OF THE BASIC COCOMO EFFORT AND SCHEDULE EQUATIONS

### The COCOMO Data Base

The Basic COCOMO estimating equations (and the other COCOMO estimating equations) have been obtained by analyzing a carefully screened sample of 63 software project data points. Table 6-5 summarizes the nature of the COCOMO data base; more detailed information is given in Chapter 29.

As can be seen from Table 6-5, the distribution of projects in the COCOMO data base is not perfectly representative of the current universe of software projects (there aren't enough COBOL data points, for example) or of the likely future universe of software projects (there aren't enough microprocessor data points, for example). However, the data base does have some representative points from all of the major sectors of the software world, and there have been no examples of data from a particular

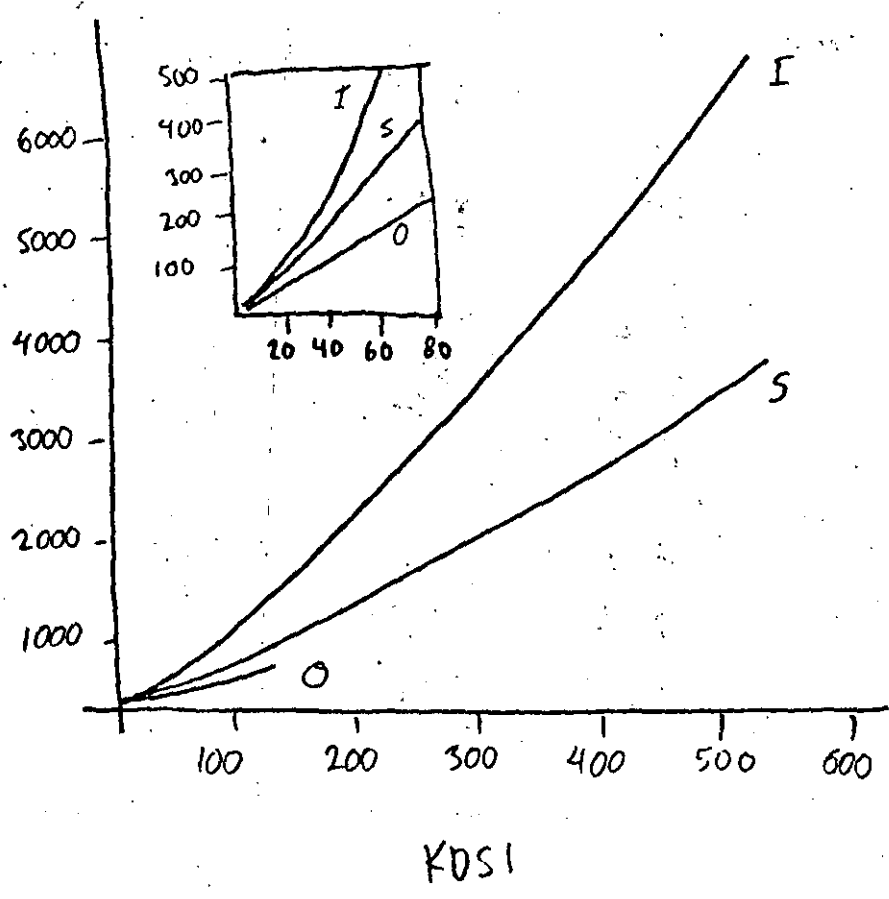
TABLE 6-5 The COCOMO Data Base

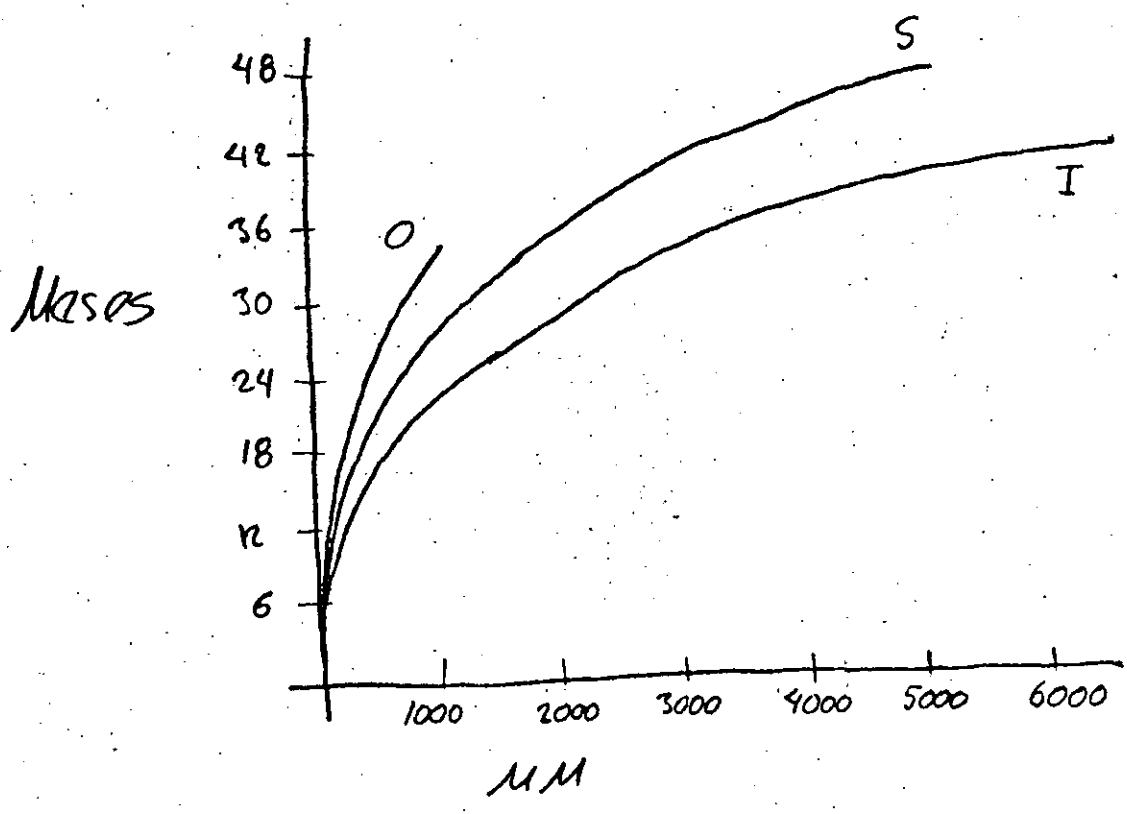
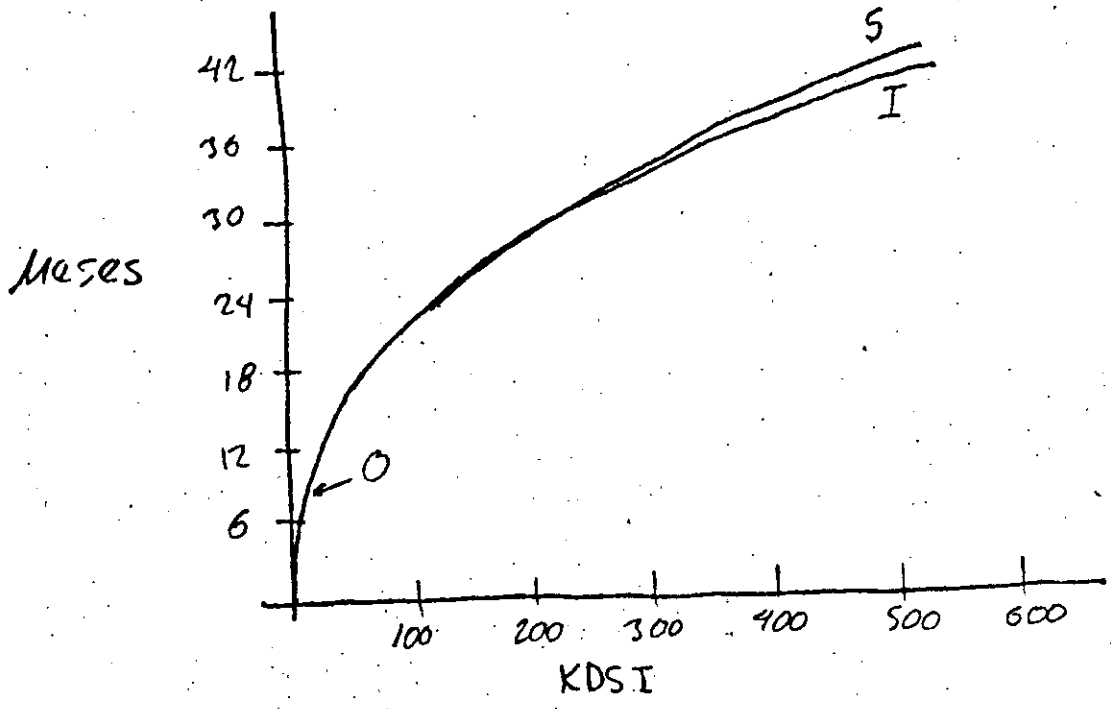
	Number of Data Points	Productivity Range (DSI/MI4)
Entire Data Base	63	20-1250
Modes: Organic	23	82-1250
Semidetached	12	41-583
Embedded	28	20-667
Types: Business	7	55-862
Control	10	20-304
Human-Machine	13	28-336
Scientific	17	47-1250
Support	8	82-583
Systems	8	28-667
Year developed: 1964-69	3	113-775
1970-74	14	20-485
1975-79	46	41-1250
Type of computer: Maxi	31	28-1250
Midi	7	114-583
Mini	21	20-723
Micro	4	41-379
Programming Languages: FORTRAN	24	28-883
COBOL	5	55-862
Jovial	5	45-583
PL/1	4	93-1250
Pascal	2	336-560
Other HOL	3	124-300
Assembly	20	20-667

# Estimaciones Basicas para Productos de Tamaño Estandar

Esfuerzo (MM)	Chico (2 KDSI)	Intermedio (8 KDSI)	Medio (32 KDSI)	Grande (128 KDSI)	Muy Grande (512 KDSI)
O	5.0	21.3	91	392	
S	6.5	31	146	687	3250
I	8.3	44	230	1216	6420
Productividad (DST/mm)					
O	400	376	352	327	
S	308	258	219	186	158
I	241	182	139	105	80
Losas					
O	4.6	8	14	24	
S	4.8	8.3	14	24	42
I	4.9	8.4	14	24	41
Personal Promedio					
O	1.1	2.7	6.5	16	
S	1.4	3.7	10	29	77
I	1.7	5.2	16	51	157

mm





# Distribución por fase de Esfuerzo

Modo Organico

Fase	Tamaño del producto				
	Pequeño (2 KDSI)	Intermedia (8 KDSI)	Mediano (32 KDSI)	Grande (128 KDSI)	Muy Grande (512 KDSI)
Esfuerzo					
Planes y Requerimientos	6%	6%	6%	6%	
Diseño del producto	16	16	16	16	
Programación	68	65	62	59	
Diseño Detallado	26	25	24	23	
Calificación y pruebas unitaria	42	40	38	36	
Integración y pruebas	16	19	22	25	
Total	100%	100%	100%	100%	
Schedule					
Planes y Requerimientos	10%	11%	12%	13%	
Diseño del producto	14	14	14	14	
Programación	63	59	55	51	
Integración y pruebas	18	22	26	30	
Total	100%	100%	100%	100%	

# Distribución por fase de Esfuerzo

# Modo Semiseparado

Fase	Tamaño del producto				
	Pequeño (2 KDSI)	Intermedio (8 KDSI)	Mediano (32 KDSI)	Grande (128 KDSI)	Muy Grande (512 KDSI)
Planes y Requerimientos	7%	7%	7%	7%	7%
Diseño del producto	17	17	17	17	17
Programación	64	61	58	55	52
Diseño Detallado	27	26	25	24	23
Modificación y prueba unitaria	37	35	33	31	29
Integración y prueba	19	22	25	28	
Total	100%	100%	100%	100%	100%

Schedule					
Planes y Requerimientos	16%	18%	20%	22%	24%
Diseño del producto	24	25	26	27	28
Programación	56	52	48	44	40
Integración y pruebas	20	23	26	29	32
Total	100%	100%	100%	100%	100%

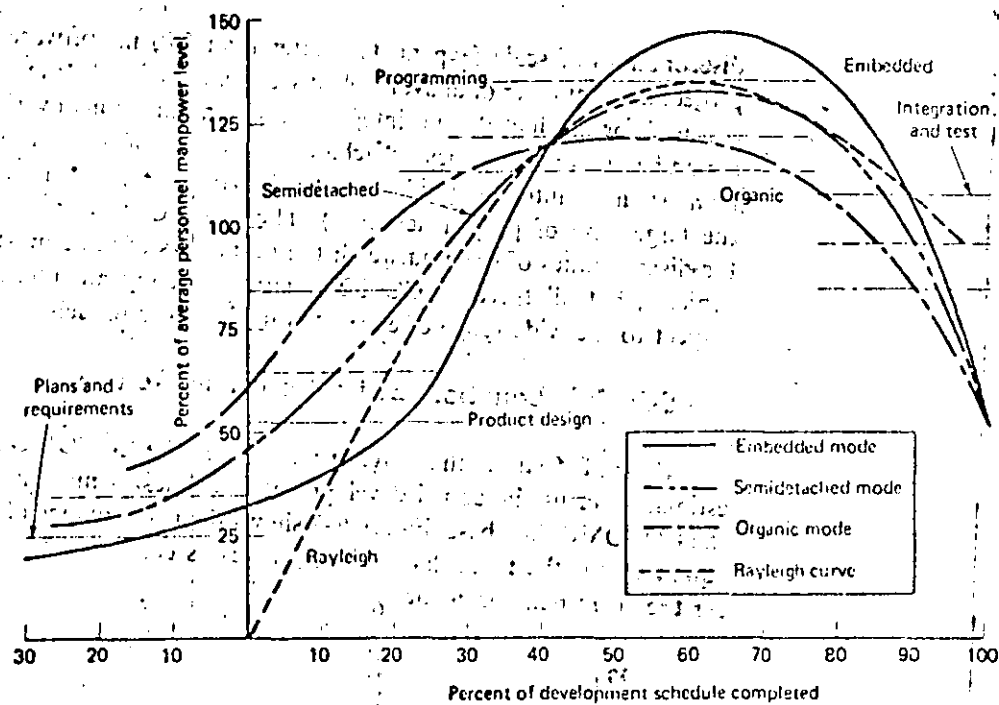


FIGURE 6-8 Basic COCOMO personnel distribution: Medium (32 KDSI) projects

whose peak effort occurs at the 60% point in the development schedule (the constant 13,300 normalizes the curve to account for the portion of the distribution left out beyond the 100% point in Fig. 6-8). Again, the Rayleigh curve is a reasonably good fit for portions of the manpower distribution (particularly for the semidetached mode), with the main exception of its zero-level behavior at the start of the project.

Thus, for practical use, we would have to tailor a portion of a Rayleigh curve to a particular mode and a particular portion of the development cycle, as we did in Chapter 5 for the organic mode. In general, it is easier and more realistic to develop a project labor plan from the average personnel per phase information given by the COCOMO model, plus as much information as you can obtain about the future availability of people to support the project and their needs for advance training, combined with your knowledge of the project's strategy for incremental development, need for early development of special support software, and so on. These topics are covered in detail in Chapter 32.

### A Final Perspective

These last points provide us with a valuable final perspective for this chapter, on the use of analytic models in project personnel planning.

*The models are just there to help, not to make your management decisions for you.*

attach  
use.  
the  
Fig.  
The  
the  
even  
in a  
6

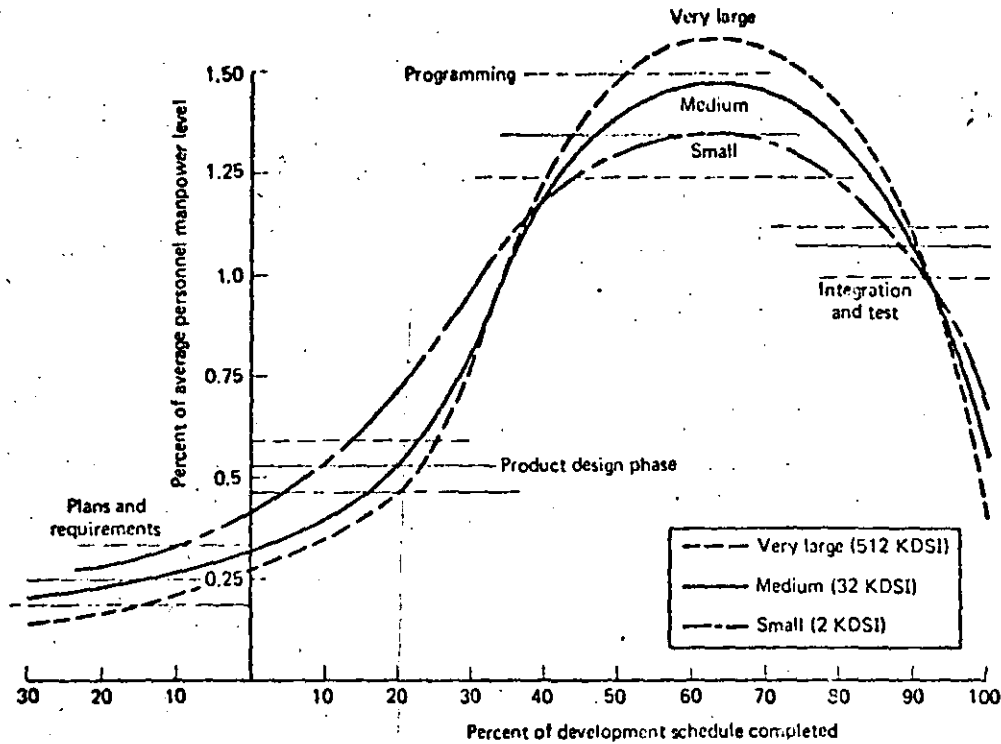


FIGURE 6-9 Basic personnel distribution: Embedded-mode projects

If you become a software project manager, and the COCOMO model or the Rayleigh distribution says you should bring another 10 people onto the project next week, while your designers say they won't be able to use any more people for another month, by all means wait another month.

### 6.6 QUESTIONS

- 6.1. The Hunt National Bank is embarking on a number of software development projects. Indicate whether each project is characteristic of the organic, semidetached or embedded mode:
- (a) A program to print various straightforward summaries of information from a daily tape of international money market transactions.
  - (b) A high-volume, real-time national electronic funds transfer system.
  - (c) A major next generation central financial management system, integrating and extending several existing programs and files.
  - (d) A model predicting near-term trends in demand for various bank services, based on existing bank transaction summary files.
  - (e) An experimental on-line transaction processing system for bank tellers.
  - (f) A simple on-line query system to support loan applications, based on the bank's loan files and a standard data base query package.





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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

MODELO BASICO  
-complemento-

Ing. Daniel Ríos Zertuche

DICIEMBRE, 1984

TABLE 7-3 Project Activity Distribution by Phase: Organic Mode

Phase	Plans and Requirements	Product Design	Programming	Integration and Test	Development	Maintenance
Product Size	S I M L	S I M L	S I M L	S I M L	S I M L	S I M L
Overall Phase Percentage	6	16	68 65 62 59	16 19 22 25		
Activity percentage						
Requirements analysis	46	15	5	3	6	7
Product design	20	40	10	6	14	13
Programming	3	14	58	34	48 47 46 45	45 44 43 42
Test planning	3	5	4	2	4	3
Verification and validation	6	6	6	34	10 11 12 13	10 11 12 13
Project office	15	11	6	7	7	7
CM/OA	2	2	6	7	5	5
Manuals	5	7	5	7	6	10

TABLE 7-2 Project Activity Distribution by Phase: Semidetached Mode

Phase	Plans and Requirements					Product Design					Programming					Integration and Test					Development					Maintenance									
	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL					
Project Size	7	7	7	7	7	17	17	17	17	17	64	61	58	55	52	19	22	25	28	31															
Overall Phase Percentage																																			
<i>Activity percentage</i>																																			
Requirements analysis	48	47	45	45	44	12.5	12.5	12.5	12.5	12.5	4	4	4	4	4	2.5	2.5	2.5	2.5	2.5	5	5	5	5	5	6.5	6.5	6.5	6	6					
Product design	16	16.5	17	17.5	18	41	41	41	41	41	8	8	8	8	8	5	5	5	5	5	13	13	13	13	13	12	12	12	12	12					
Programming	2.5	3.5	4.5	5.5	6.5	12	12.5	13	13.5	14	58.5	58.5	58.5	58.5	58.5	32	35	37	39	41	45	45	44.5	44.5	44.5	41.5	41.5	41	41	41					
Test planning	2.5	3	3.5	4	4.5	4.5	5	5.5	6	6.5	4	4.5	5	5.5	6	2.5	2.5	3	3	3.5	4	4	4.5	5	5.5	3	3	3.5	4	4.5					
Verification and validation	6	6.5	7	7.5	8	6	6.5	7	7.5	8	7	7.5	8	8.5	9	32	31	29.5	28.5	27	11	12	13	13.5	14	11	12	13	13.5	14					
Project office	15.5	14.5	13.5	12.5	11.5	13	12	11	10	9	7.5	7	6.5	6	5.5	8.5	8	7.5	7	6.5	8.5	8	7.5	7	6.5	8.5	8	7.5	7	6.5					
CM/CA	3.5	3	3	3	2.5	3	2.5	2.5	2.5	2	7	6.5	6.5	6.5	6	8.5	8	8	8	7.5	6.5	6	6	6	5.5	6.5	6	6	6	5.5					
Manuals	6	6	5.5	5	5	8	8	7.5	7	7	6	6	5.5	5	5	8	8	7.5	7	7	7	7	6.6	6	6	11	11	10.5	10.5	10.5					

TABLE 7-1 Project Activity Distribution by Phase: Embedded Mode

Phase Product Size	Plans and Requirements					Product Design					Programming					Integration and Test					Development					Maintenance									
	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL					
Overall Phase Percentage	8	8	8	8	8	12	12	12	12	12	60	57	54	51	49	22	25	23	21	24															
Activity percentage																																			
Requirements analysis	50	48	46	44	42	10	10	10	10	10	3	3	3	3	3	2	2	2	2	2	4	4	4	4	4	6	6	6	5	5					
Product design	12	13	14	15	16	42	42	42	42	42	6	6	6	6	6	4	4	4	4	4	12	12	12	12	12	11	11	11	11	11					
Programming	2	4	6	8	10	10	11	12	13	14	55	55	55	55	55	32	36	40	44	48	42	43	43	44	45	38	39	39	40	41					
Test planning	2	3	4	5	6	4	5	6	7	8	4	5	6	7	8	3	3	4	4	5	4	4	5	5	7	3	3	4	5	6					
Verification and validation	6	7	8	9	10	6	7	8	9	10	8	9	10	11	12	30	28	25	23	20	12	13	14	14	14	12	13	14	14	14					
Project office	15	14	12	10	8	15	13	11	9	7	9	8	7	6	5	10	9	8	7	6	10	9	8	7	6	10	9	8	7	5					
CM/GA	5	4	4	4	3	4	3	3	3	2	8	7	7	7	6	10	9	9	9	8	8	7	7	7	6	8	7	7	7	6					
Manuals	7	7	6	5	5	9	9	8	7	7	7	7	6	5	5	9	9	8	7	7	8	8	7	6	6	12	12	11	11	11					

## ESTRUCTURA DEL PROYECTO

Un organigrama nos dice como el gerente del proyecto ha delegado autoridad y responsabilidad por funciones del proyecto a la gente identificada en la gráfica.

Es valiosa al clasificar las responsabilidades del proyecto y en acuerdo con principios de buena administración del proyecto (tales como "unidad de comando" y "paridad de autoridad y responsabilidad").

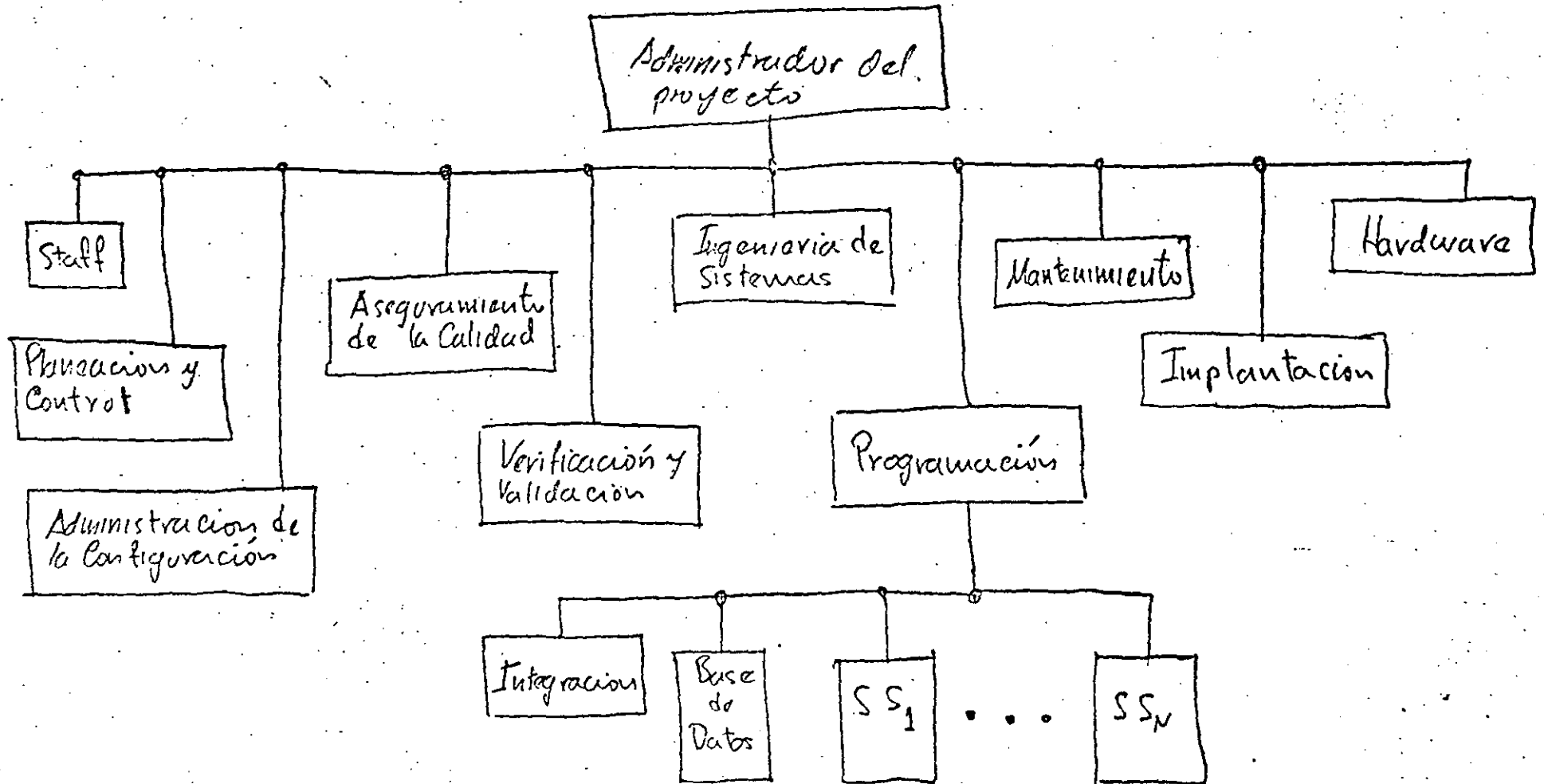
### Lineamientos para la elaboración del organigrama

- 1.- Combine funciones adyacentes en el organigrama generalizado.
- 2.- Combine una función con menos de 2 personas con la vecina a menos que:
  - a) representa el gerente o un conjunto de funciones generalizadas.
  - b) si cuenta con el menos 0.5 personas y creciera a una función completa en la siguiente fase.
- 3.- Divida una función en su gerente y un conjunto de funciones subordinadas si cuenta con más de 7 personas.

4.- Mantenga el ambito de control (número de funciones administrativas) de cualquier gerente a no más de 7 personas.

5.- Si cualquiera de los lineamientos presentan conflictos con el sentido común, olvide el lineamiento y use el sentido común.

# Organigrama Generalizado del proyecto de Software



FUNCIONES DE LA OFICINA DEL PROYECTO

Gerencia del Proyecto

Staff

Planeación y Control

INGENIERIA DE SISTEMAS

Análisis de Requerimientos

Diseño del producto

Manuales

VALIDACION Y VERIFICACION

Validación

Verificación

Pruebas

SS<sub>1</sub> ... SS<sub>n</sub>

Programación de los subsistemas de  
software



8<sup>3</sup>

IMPLANTACION

Conversión

Instalación

Entrenamiento

Actitudes relacionadas con la fase de  
implantación

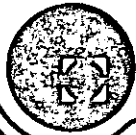
## Limitaciones del Modelo Básico

Una de las limitaciones del Modelo Básico ( y de todos los modelos de distribución por fase a la fecha) es que no representa " formas altamente secuenciales de desarrollo incremental". Las distribuciones de esfuerzo ajustan bastante bien, pero el cálculo del schedule debe calcularse diferente.

El Modelo Básico calcula el nivel promedio de personal en cada fase.

La limitación principal del modelo básico es que este no incorpora el efecto de ningún factor de costo además del de instrucciones fuente entregadas, y el tráfico de cambio anual para mantenimiento.

Con respecto a la Base de Datos el modelo Básico nos da estimaciones con un factor de 1.3 con respecto a la realidad el 29% del tiempo y dentro de un factor de 2 únicamente el 60% del tiempo.



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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

MODELO INTERMEDIO

Ing. Daniel Ríos Zertuche

DICIEMBRE, 1984

## MODELO INTERMEDIO

El modelo intermedio incorpora 15 variables predictoras más las cuales toman en cuenta gran parte de la variación de costos del proyecto de software que quedan sin explicación en el modelo Básico.

Con respecto a la Base de Datos las estimaciones del modelo intermedio están dentro del 20% de los datos actuales el 68% del tiempo.

### + Atributos del producto

RELY    Confiabilidad requerida del producto

DATA    Tamaño de la Base de Datos

CPLX    Complejidad del producto

### + Atributos de la Computadora

TIME    Restricciones de tiempo de ejecución

STOR    Restricciones de almacenamiento principal

VIRT    Volatilidad de la Máquina Virtual

TURN    Tiempo de respuesta de la Computadora

### + Atributos de Personal

ACAP    Capacidad de los Analistas

AEXP    Experiencia en las Explicaciones

PCAP    Capacidad de los Programadores

VEXP Experiencia en la Máquina Virtual  
LEXP Experiencia en el Lenguaje de Programación

+ Atributos del Proyecto

MODP Prácticas Modernas de Programación  
TOOL Uso de Herramientas de Software  
SCED Schedule requerido de Desarrollo

Cada uno de estos atributos determina un factor multiplicativo el cual estima el efecto del atributo en el esfuerzo de desarrollo de software. Estos multiplicadores se aplican a las estimaciones nominales de esfuerzo de desarrollo para obtener un estimado refinado del esfuerzo de desarrollo de software. Un proceso similar se lleva a cabo para determinar un estimado refinado del esfuerzo de mantenimiento de software.

Estimación del Nominal del esfuerzo

Modo de Desarrollo	Ecuación de Esfuerzo Nominal
Orgánico	$(MM)_{nom} = 3.2(KDSI)^{1.05}$
Semiseparado	$(MM)_{nom} = 3.0(KDSI)^{1.12}$
Incrustado	$(MM)_{nom} = 2.8(KDSI)^{1.20}$

## Distribución por fase y actividad del Esfuerzo y Schedule.

El Modelo intermedio usa las mismas relaciones de estimación para el schedule de desarrollo y la distribución de actividades que las usadas en el Modelo Básico.

Esto es:

- + El schedule de desarrollo estimado, TDEV, es calculado del esfuerzo de desarrollo estimado en el modelo intermedio.
- + La distribución del porcentaje de esfuerzo y schedule por fase son obtenidos como una función del modo y tamaño del producto.
- + La distribución del porcentaje de esfuerzo por actividad y fase es obtenido como una función del modo y tamaño del producto.

## Análisis de Sensibilidad

El modelo intermedio nos permite realizar un análisis de sensibilidad con respecto a los factores de costo, el cual nos permite estimar el efecto de cambios en estos factores en el costo del software.

TABLE 8-2 Software Development Effort Multipliers

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
<b>Product Attributes</b>						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.09	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
<b>Computer Attributes</b>						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
<b>Personnel Attributes</b>						
ACAP Analyst capability	1.46	1.10	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PGAP Programmer capability	1.42	1.17	1.00	.85	.70	
VEXP Virtual machine experience*	1.21	1.10	1.00	.90		
LEXP Programming language experience*	1.14	1.07	1.00	.95		
<b>Project Attributes</b>						
MCDP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

\*For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

TABLE 8-3 Software Cost Driver Ratings

Ratings



TABLE 8-3 Software Cost Driver Ratings

Cost Driver	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
<b>Product attributes</b>						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{DB \text{ bytes}}{\text{Prog. DS}} < 10$	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} > 1000$	
CPLX	See Table 8-4					
<b>Computer attributes</b>						
TIME			$\leq 50\%$ use of available execution time	70%	85%	95%
STOR			$\leq 50\%$ use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround < 4 hours	4-12 hours	> 12 hours	
<b>Personnel attributes</b>						
ACAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
AEXP	$\leq 4$ months experience	1 year	3 years	5 years	12 years	
PCAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
VEXP	$\leq 1$ month experience	4 months	1 year	3 years		
LEXP	$\leq 1$ month experience	4 months	1 year	3 years		
<b>Project attributes</b>						
MOOP	No use	Beginning use	Some use	General use	Routine use	
TOOL	Basic microprocessor tools	Basic mini tools	Basic mid/maxi tools	Strong maxi programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

\* Team rating criteria: analysis (programming) ability, efficiency, ability to communicate and cooperate

TABLE 8-4 Module Complexity Ratings versus Type of Module

Rating	Control Operations	Computational Operations	Device dependent Operations	Data Management Operations	TABLE Rating
Very low	Straightline code with a few non-nested SP* operators: COs, CASEs, IFTHENCEs. Simple predicates	Evaluation of simple expressions: e.g., $A = B + C$ * ( $D - E$ )	Simple read, write statements with simple formats	Simple arrays in main memory	Very low
Low	Straightforward nesting of SP operators. Mostly simple predicates	Evaluation of moderate-level expressions, e.g., $D = SORT (B**2-4.*A*C)$	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap	Single file subsetting with no data structure changes, no edits, no intermediate files	Low
Nominal	Mostly simple nesting. Some inter-module control. Decision tables	Use of standard math and statistical routines. Basic matrix/vector operations	I/O processing includes device selection, status checking and error processing	Multi-file input and single file output. Simple structural changes, simple edits	Nominal
High	Highly nested SP operators with many compound predicates. Ocuco and stack control. Considerable inter-module control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns	Operations at physical I/O level (physical storage address translations; seeks, reads, etc); Optimized I/O overlap	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level	High
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling	Difficult but structured N.A.; near-singular matrix equations, partial differential equations	Routines for interrupt diagnosis, servicing, masking. Communication line handling	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization	Very high
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control	Difficult and unstructured N.A.; highly accurate analysis of noisy, stochastic data	Device timing-dependent coding, micro-programmed operations	Highly coupled, dynamic relational structures. Natural language data management	

\* SP = structured programming

**TABLE B-5** Project Activity Differences due to Required Software Reliability (acronyms are explained in Appendix C)

Rating	Rqts. and Product Design	Detailed Design	Code and Unit Test	Integration and Test
<b>Very low</b>	Little detail Many TBDs Little verification Minimal OA, CM, draft user manual, test plans Minimal PDR	Basic design information Minimal OA, CM, draft user manual, test plans Informal design inspections	No test procedures Minimal path test, standards check Minimal OA, CM Minimal I/O and off-nominal tests Minimal user manual	No test procedures Many requirements untested Minimal OA, CM Minimal stress, off-nominal tests Minimal as-built documentation
<b>Low</b>	Basic information, verification Frequent TBDs Basic OA, CM, standards, draft user manual, test plans	Moderate detail Basic OA, CM, draft user manual, test plans	Minimal test procedures Partial path test, standards check Basic OA, CM, user manual Partial I/O and off-nominal tests	Minimal test procedures Frequent requirements untested Basic OA, CM, user manual Partial stress, off-nominal tests
<b>Nominal</b>	Nominal project V & V			
<b>High</b>	Detailed verification, OA, CM, standards, PDR, documentation Detailed test plans, procedures	Detailed verification, OA, CM, standards, CDR, documentation Detailed test plans, procedures	Detailed test procedures, OA, CM, documentation Extensive off-nominal tests	Detailed test procedures, OA, CM, documentation Extensive stress, off-nominal tests
<b>Very high</b>	Detailed verification, OA, CM, standards, PDR, documentation IV & V interface Very detailed test plans, procedures	Detailed verification, OA, CM, standards, CDR, documentation Very thorough design inspections Very detailed test plans, procedures IV & V interface	Detailed test procedures, OA, CM, documentation Very thorough code inspections Very extensive off-nominal tests IV & V interface	Very detailed test procedures, OA, CM, documentation Very extensive stress, off-nominal tests IV & V interface

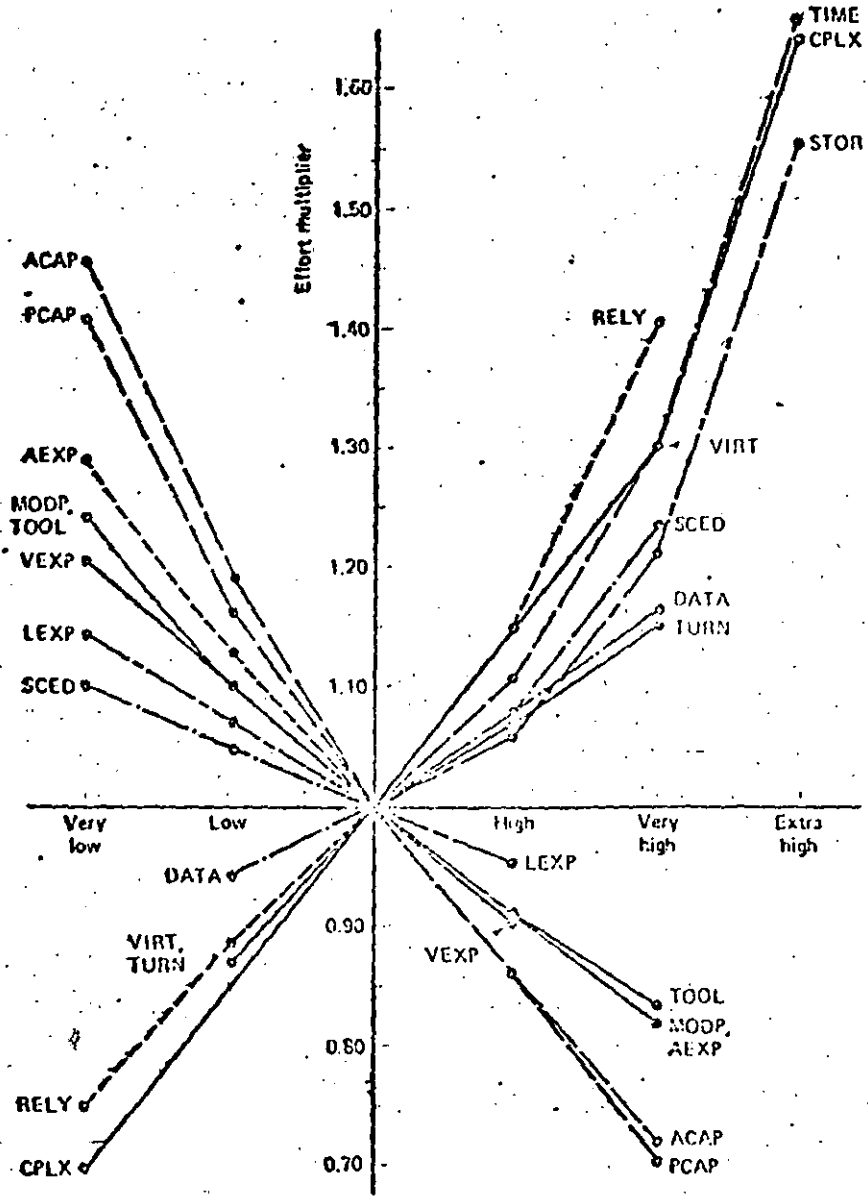


FIGURE 8-2 Intermediate COCOMO effort multipliers

- The estimated development schedule, TDEV, is calculated from the Intermediate COCOMO estimate of development effort in MM, using the equations in Table 6-1.
- The percentage distribution of effort and schedule by phase is obtained as a function of mode and product size, using Table 6-8.
- The percentage distribution of effort by activity and phase is obtained as a function of mode and product size, using Tables 7-1 through 7-3.

## Ajuste a la estimación del esfuerzo anual de mantenimiento

Los Multiplicadores de esfuerzo en el Modelo Intermedio pueden ser aplicados a la fase de mantenimiento.

SCE0 (schedule de desarrollo requerido) no tiene sentido en la fase de mantenimiento.

RELY (confiabilidad requerida del producto) debido a la diferencia en impacto de este factor en el desarrollo y mantenimiento este factor cambio, sin embargo se debe usar el mismo nivel tanto en el desarrollo como en el mantenimiento.

Multiplicador de esfuerzo de mantenimiento Rely

Muy bajo	Bajo	Nominal	Alto	Muy alto
1.35	1.15	1.0	0.98	1.10

Lo anterior representa dos efectos:

- 1.- Mientras menor es la confiabilidad requerida, menor esfuerzo es requerido para mantener el nivel requerido.
- 2.- Mientras menor la confiabilidad requerida; mayor es el esfuerzo que se requiere para fijar los errores latentes en

el software, y para actualizar un producto de software -  
con documentación y código inadecuado.

### MOOP (Prácticas de Programación Modernas)

El efecto de las Prácticas de Programación Modernas durante -  
el desarrollo tiene dos efectos en el nivel requerido de man-  
tenimiento de software.

- 1.- Mientras más PMP son usadas, mayor será el ahorro en el -  
esfuerzo de mantenimiento.
- 2.- Mientras más PMP son usadas, más fácil es mantener produc-  
tos grandes con la misma eficiencia que productos peque-  
ños.

### Multiplicadores de Esfuerzo de Mantenimiento

Tamaño del Producto (KDSI)	N i v e l				
	Muy bajo	Bajo	Nominal	Alto	Muy alto
2	1.25	1.12	1.0	0.90	0.81
8	1.30	1.14	1.0	0.88	0.77
32	1.35	1.16	1.0	0.86	0.74
128	1.40	1.18	1.0	0.85	0.72
512	1.45	1.20	1.0	0.84	0.70

Estimación de los Efectos de Adaptar Software Existente

El aprovechamiento de software existente puede requerir un gran esfuerzo en:

- 1.- Rediseño del software adaptado para alcanzar los objetivos del nuevo producto.
- 2.- Rehacer porciones del código para acomodar características rediseñadas o cambios en el nuevo medio ambiente del producto.
- 3.- Integración del código adaptado en el nuevo medio ambiente del producto y pruebas del producto de software resultante.

Ecuaciones para Estimación de Adaptaciones

Los efectos del software adaptado son manejados en el Modelo - calculando el número equivalente de instrucciones fuente entregadas. (EDSI), el cual es usado en lugar de las DSI en las relaciones de estimación del Modelo.

EDSI es calculada de las siguientes cantidades de adaptación estimadas:

ADSI DSI Adaptadas . El número de instrucciones fuente entregadas adaptadas del software existente para formar el nuevo producto.

DM Porcentaje de diseño Modificado. El porcentaje del diseño de software adaptado, el cual es modificado en orden para adaptarlo a los nuevos objetivos y el medio ambiente. (Esta es necesariamente una cantidad subjetiva) .

IM Porcentaje de Integración Requerido para el Software Modificado. El porcentaje de esfuerzo requerido para integrar el software adaptado dentro de un producto y para probar el producto resultante comparado con la cantidad normal de integración y esfuerzo de pruebas para el software de tamaño comparable.



Las ecuaciones para calcular el EDSI involucran una cantidad intermedia AAF el factor de ajuste de adaptación

$$AAF = 0.40(DM) + 0.30(CM) + 0.30(IM)$$

$$EDSI = (ADSI) \frac{AAF}{100}$$

ESTIMACION A NIVEL DE COMPONENTE

Forma de Estimación a Nivel de Componente (CLEF)  
(Component Level Estimation Form)

Procedimiento para el uso de la forma.

- 1.- Identifique todas las componentes del producto de software en la columna 1.
- 2.- Estime el tamaño en DSI de todas las componentes.  
Si el componente no es adaptado de software existente, coloque su tamaño en la columna 2 (EDSI). Si es adaptado, compute su factor de ajuste de adaptación (AAF) por la ecuación dada abajo, coloquelo en la columna 5, entonces calcule las DSI equivalentes (EDSI) y coloquelo en la columna 2.
- 3.- Sume el total de EDSI para el producto y coloquelo en el renglón 11 columna 2
- 4.- Use la ecuación apropiada de esfuerzo nominal para el modo de desarrollo especificado (dado abajo) para estimar el total nominal de esfuerzo de desarrollo (MM)nom como una función de las EDSI y coloquelo en el renglón 12 columna 2.
- 5.- Calcule la productividad nominal  
$$(EDSI/MM)nom=(totalEDSI)/(MMnom)$$
 y coloquelo en el renglón 13, columna 2

6.- Para cada componente, calcule

$$(MM)_{nom} = EDSI / (EDSI/MM)_{nom}$$

y coloquelo en la columna 20

7.- Provea los factores de costo (columnas 4 a 18) para todos los compentes, usando las escalas de los niveles.

8.- Coloque los multiplicadores de esfuerzo correspondientes para todos los elementos en las columnas 4 a 18.

9.- Para cada componente (renglón), calcule el factor de ajuste de esfuerzo (EAF) como el producto de los multiplicadores de esfuerzo en las columnas 4 a 18 y coloquelo en la columna 19.

10.- Multiplique (MM) nom (columna 20) para cada componente por su EAF para producir el estimado ajustado para  $(MM)_{DEV}$ , el cual se coloca en la columna 21.

11.- Agregue el total de estimados hombres-mes ajustados para todos los componentes y coloquelos en el renglón 11 columna 21.

12.- Use la ecuación requerida básica de schedule de desarrollo para el modo de desarrollo especificado y coloquela en el renglón 12, columna 21

13.- Para cada componente y para el producto completo, calcule la productividad estimada

$$EDSI/MM = EDSI/MM_{DEV}$$

y colóquela en la columna 22

14.- Estime el costo promedio de la mano de obra (\$K/MM) para cada componente y colóquela en la columna 25

15.- Calcule el costo en pesos para cada componente

$$\$K = (MM_{DEV}) (\$K/MM)$$

y colóquelo en la columna 23 (mitad inferior)

16.- Agregue el costo de desarrollo del producto en \$K y colóquelo en el renglón 11, columna 23

17.- Para cada componente y para el producto completo, calcule el costo por instrucción

$$$/EDSI = (1000) (\$K) / EDSI$$

y colóquelo en la columna 24

Procedimiento para usar la forma CLEF para la estimación de los Costos de Mantenimiento del Software.

- 1.- Para cada componente, del factor de ajuste de esfuerzo calculado para el desarrollo de software en la parte baja de la columna 19
- 2.- Para cada componente, identifique cualquier cambio en los factores de costo entre el mantenimiento y desarrollo colocando el multiplicador de desarrollo en la parte baja de la columna del factor-costo y el correspondiente multiplicador de mantenimiento en la mitad superior de la columna de factor de costo (columnas 4 a 18).

Los multiplicadores de esfuerzo pueden cambiar por cualquiera de las siguientes razones:

- + Multiplicadores fuera del nominal para SCED deben cambiarse a 1 para el mantenimiento.
- + Multiplicadores no nominales para RELY y MODP para desarrollo tendrán diferente valor para mantenimiento.
- + El nivel de ciertos factores puede cambiar (por ejemplo, la experiencia).

- 3.- Para cada componente, calcule su mantenimiento EAF

$$(EAF)_m = (EAF)_{dev} \frac{\text{Producto de los multiplicadores de mantenimiento combinados}}{\text{Producto de los multiplicadores de desarrollo combinados}}$$

y colocados en la parte superior de la columna 19.

4.- Para cada componente del tráfico de cambio anual (ACT) como una fracción (tal como 0.10 para 10%) en la columna 22

5.- Si todos los componentes tienen una AAF de 1.0, coloque los hombres-mes de desarrollo nominal para cada componente en la columna 20.

Si no, necesitamos recalcar los (MM)nom basados en el tamaño del producto a ser mantenido. Para cada componente cuyo AAF no es 1.0 calcule su actual

$$DSI = \frac{EDSI}{AAF}$$

y coloquelo en la columna 2. Entonces

+ Calcule un DSI total revisado y coloquelo en el renglón 11 columna 2.

+ Calcule un (MM)nom revisado para el producto usando el total de DSI revisado y la ecuación apropiada de esfuerzo nominal para el modo dado y coloquelo en el renglón 12, columna 2.

+ Para cada componente, calcule un (MM)nom revisado

$$(MM)nom = \frac{DSI}{(DSI/MM)nom}$$

21  
y coloquelo en la columna 21.

7.- Agregue el total de esfuerzo de mantenimiento para todos los componentes y coloquelos en el renglón 11 columna 21

8.- Estime el costo promedio de trabajo (\$K/MM) para mantener cada componente y coloquelo en la parte superior de la columna 23.

9.- Calcule el costo anual en pesos del mantenimiento de cada componente

$$\$K = (MM)_{AM} (\$K/MM)$$

y coloquelo en la columna 23

10.- Agregue el costo total del mantenimiento anual para todos los componentes y coloquelos en el renglón 11, columna 23.

1	2	Product				Computer				Personnel attrib.				Project				23	24				
		3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18			19	20	21	22
COMPONENT	ECSI	AAF	RELY DATA	CPLX	TIME	STOR	VIRT	TURN	ACAP	AEXP	PCAP	VEXP	LEXP	VOOP	TODL	SCED	EAF	MM NOM	MM DEN AN	ECSI MM ACT	'K	ECSI	
1.																							
2.																							
3.																							
4.																							
5.																							
6.																							
7.																							
8.																							
9.																							
10.																							
11.		TOTAL ECSI															Totals						
12.		(MM) <sub>NOM</sub>															Development mode: _____		Schedule (Mo)				
13.		ECSI (MM) <sub>NOM</sub>																					

COCOMO software cost model: Component level estimating form (CLEF)

Intermediate COCOMO Component-Level Estimation Form (CLEF)





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

FACTORES ECONOMICOS DEL DESARROLLO DE SOFTWARE

IEEE TRANSACTIONS ON SOFTWARE  
ENGINEERING

(anexo págs. 13 y 14)

DICIEMBRE 1984

provide his own fraction), and uses its standard techniques to estimate the resulting life-cycle effort distribution.

**Maintenance:** The estimator provides a parameter indicating the quality level of the developed code. PRICE SL uses it to estimate the effort required to eliminate remaining errors.

### The COConstructive COst Model (COCOMO) [11]

The primary motivation for the COCOMO model has been to help people understand the cost consequences of the decisions they will make in commissioning, developing, and supporting a software product. Besides providing a software cost estimation capability, COCOMO therefore provides a great deal of material which explains exactly what costs the model is estimating, and why it comes up with the estimates it does. Further, it provides capabilities for sensitivity analysis and tradeoff analysis of many of the common software engineering decision issues.

COCOMO is actually a hierarchy of three increasingly detailed models which range from a single macro-estimation scaling model as a function of product size to a micro-estimation model with a three-level work breakdown structure and a set of phase-sensitive multipliers for each cost driver attribute. To provide a reasonably concise example of a current state of the art cost estimation model, the intermediate level of COCOMO is described below.

Intermediate COCOMO estimates the cost of a proposed software product in the following way.

1) A nominal development effort is estimated as a function of the product's size in delivered source instructions in thousands (KDSI) and the project's development mode.

2) A set of effort multipliers are determined from the product's ratings on a set of 15 cost driver attributes.

3) The estimated development effort is obtained by multiplying the nominal effort estimate by all of the product's effort multipliers.

4) Additional factors can be used to determine dollar costs, development schedules, phase and activity distributions, computer costs, annual maintenance costs, and other elements from the development effort estimate.

**Step 1—Nominal Effort Estimation:** First, Table IV is used to determine the project's development mode. Organic-mode projects typically come from stable, familiar, forgiving, relatively unconstrained environments, and were found in the COCOMO data analysis of 63 projects have a different scaling equation from the more ambitious, unfamiliar, unforgiving, tightly constrained embedded mode. The resulting scaling equations for each mode are given in Table V; these are used to determine the nominal development effort for the project in man-months as a function of the project's size in KDSI and the project's development mode.

For example, suppose we are estimating the cost to develop the microprocessor-based communications processing software for a highly ambitious new electronic funds transfer network with high reliability, performance, development schedule, and interface requirements. From Table IV, we determine that these characteristics best fit the profile of an embedded-mode project.

We next estimate the size of the product as 10 000 delivered

TABLE IV  
COCOMO SOFTWARE DEVELOPMENT MODES

Feature	Mode		
	Organic	Semidetached	Embedded
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	100-300 KDSI	All sizes
Examples	Batch data reduction Scientific models Business models Familiar OS, compiler Simple inventory, production control	Most transaction processing systems New OS, DBMS Ambitious inventory, production control Simple command control	Large, complex transaction processing systems Ambitious, very large OS Avionics Ambitious command control

TABLE V  
COCOMO NOMINAL EFFORT AND SCHEDULE EQUATIONS

DEVELOPMENT MODE	NOMINAL EFFORT	SCHEDULE
Organic	$(MM)_{NOI} = 3.2(KDSI)^{1.05}$	$TDEV = 2.5(MM_{DEV})^{0.38}$
Semidetached	$(MM)_{NOI} = 3.0(KDSI)^{1.12}$	$TDEV = 2.5(MM_{DEV})^{0.35}$
Embedded	$(MM)_{NOI} = 2.8(KDSI)^{1.20}$	$TDEV = 2.5(MM_{DEV})^{0.32}$

(KDSI = thousands of delivered source instructions)

source instructions, or 10 KDSI. From Table V, we then determine that the nominal development effort for this Embedded-mode project is

$$2.8(10)^{1.20} = 44 \text{ man-months (MM)}$$

**Step 2—Determine Effort Multipliers:** Each of the 15 cost driver attributes in COCOMO has a rating scale and a set of effort multipliers which indicate by how much the nominal effort estimate must be multiplied to account for the project's having to work at its rating level for the attribute.

These cost driver attributes and their corresponding effort multipliers are shown in Table VI. The summary rating scales for each cost driver attribute are shown in Table VII, except for the complexity rating scale which is shown in Table VIII (expanded rating scales for the other attributes are provided in [11]).

The results of applying these tables to our microprocessor communications software example are shown in Table IX. The effect of a software fault in the electronic fund transfer system could be a serious financial loss; therefore, the project's RELY rating from Table VII is High. Then, from Table VI, the effort multiplier for achieving a High level of required reliability is 1.15, or 15 percent more effort than it would take to develop the software to a nominal level of required reliability.

**TABLE VI**  
**INTERMEDIATE COCOMO SOFTWARE DEVELOPMENT EFFORT MULTIPLIERS**

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
<b>Product Attributes</b>						
RELY Required software reliability	.75	.86	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.18	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
<b>Computer Attributes</b>						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility*		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
<b>Personnel Attributes</b>						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.28	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.85		
<b>Project Attributes</b>						
MOOP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

\* For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

**TABLE VII**  
**COCOMO SOFTWARE COST DRIVER RATINGS**

Cost Driver	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
<b>Product attributes</b>						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		DB bytes < 10 Prog DSI < 10	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} \geq 1000$	
CPLX	See Table 8					
<b>Computer attributes</b>						
TIME			≤ 50% use of available execution time	70%	85%	95%
STOR			≤ 50% use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround < 4 hours	4-12 hours	> 12 hours	
<b>Personnel attributes</b>						
ACAP	15th percentile	25th percentile	55th percentile	75th percentile	90th percentile	
AEXP	< 4 months experience	1 year	3 years	6 years	12 years	
PCAP	15th percentile	25th percentile	55th percentile	75th percentile	90th percentile	
VEXP	< 1 month experience	4 months	1 year	3 years		
LEXP	< 1 month experience	4 months	1 year	3 years		
<b>Project attributes</b>						
MOOP	No use	Beginning use	Some use	General use	Routine use	
TOOL	No use	Basic microprocessor tools	Basic mid/main tools	Strong main programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

\* Team rating criteria: analysis, programming ability, efficiency, ability to communicate and cooperate.

DIRECTORIO DE ALUMNOS DEL CURSO "FACTORES ECONOMICOS DE DESARROLLO DE SOFTWARE" IMPARTIDO EN ESTA DIVISION DEL 3 AL 8 DE DICIEMBRE 1984.

1.- ALVARADO VAZQUEZ MOISES  
U. A. E. M.  
PROGRAMADOR  
AV. UNIVERSIDAD No. 101  
COL. CHAMILPA  
CUERNAVACA, MORELOS  
13-26-44

PRIV. E. ZAPATA No. 99  
CUERNAVACA, MOR.

2.- ANDRADE RUIZ HECTOR  
DIREC. GRAL. CORREOS  
JEFE DE OFICINA  
ALDAMA No. 218-50. PISO  
COL. BUENAVISTA  
DELEGACION VENUSTIANO CARRANZA  
526-87-73

STA. BARBARA No. 37  
COL. MOLINO DE STO. DOMINGO  
DELEGACION ALVARO OBREGON

3.- ARMENTA CRUZ JESUS MOISES  
DIREC. GRAL. CONCESIONES Y PERMISOS  
DE TELECOMUNICACIONES  
COORDINADOR DE TECNICOS ESPECIALIZADOS  
AV. POPOCATEPETL No. 506-B  
COL. XOCO  
DELEGACION BENITO JUAREZ  
03330 MEXICO, D.F.  
688-98-53

SUR 67 No. 3125-2  
COL. VIADUCTO PIEDAD  
DELEGACION IXTACALCO  
08200 MEXICO, D.F.  
530-23-65

4.- BORAU GARCIA JORGE  
ATISA ATKINS, S.A. DE C. V.  
GERENTE DE INFORMATICA  
BAHIA DE CORRIENTES No. 77  
COL. VERONICA ANZURES  
DELEGACION MIGUEL HIDALGO  
11300 MEXICO, D.F.  
250-82-11

COLINA DE LAS VENTISCAS No. 37  
COL. BULEVARES EDO. DE MEXICO  
53140 MEXICO, D.F.  
560-56-30

5.- BYRD NERI SERGIO A.  
CENTRO DE CALCULO F. I.  
SUBCOORDINADOR PROYECTOS ADMOS.  
CIUDAD UNIVERSITARIA  
DELEGACION COYOACAN  
550-57-34

CALLE DILIGENCIAS No. 153 E-504  
COL. SAN PEDRO MARTIN  
DELEGACION TLALPAN  
14650 MEXICO, D.F.  
550-52-34

6.- CORRAL GARCIA HUMBERTO  
S. A. R. H.  
JEFE DE OFICINA  
REFORMA No. 107-100. PISO  
546-59-04

AV. UNIVERSIDAD No. 1900 ED. 16-301  
COL. OXTOPULCO COYOACAN  
04310 MEXICO, D.F.  
658-26-32

7.- CRUZ CEBALLOS VICTOR ARMANDO  
S. C. T.  
ANALISTA DE SISTEMAS

AV. ALEZ No. 290 COL. PUEBLA  
DELEGACION VENUSTIANO CARRANZA  
15020 MEXICO, D.F.  
763-46-13



- 8.- ESCUTIA ACOSTA RAUL  
S. A. R. H.  
PROGRAMADOR  
AV. INSURGENTES CENTRO No. 30  
COL. JUAREZ  
DELEGACION CUAUHTEMOC  
06000 MEXICO, D.F.  
591-18-35
- 9.- FIGUEROA MARQUEZ YOLANDA  
U. N. A. M. F. I.  
TECNICO ACADEMICO ASOCIADO "C"  
AV. UNIVERSIDAD No. 1050  
550-52-15 ext. 2143
- 10.- FONSECA CAMPOS ARTURO  
D. D. F.
- 11.- GARCIA CORELLA JAIME  
PROGRAMA UNIVERSITARIO DE COMPUTO  
TECNICO AUXILIAR "C"  
U. N. A. M.
- 12.- GARCIA MORALES PABLO ALFONSO  
ACEROS FORTUNA, S.A.  
ANALISTA PROGRAMADOR  
AV. LIC. JUAN FERNANDEZ ALBARRAN 31  
COL. ZONA INDUSTRIAL SAN PABLO XALPA  
TLALNEPANTLA EDO. DE MEXICO
- 13.- GRANDOS REYES JOSE LUIS  
ACEROS FORTUNA, S.A. DE C.V.  
JEFE DEL DEPTO. DE SISTEMAS  
AV. LIC. JUAN FERNANDEZ ALBARRAN  
COL. SAN PABLO XALPA  
DELEGACION TLALNEPANTLA EDO. MEX.  
392-50-00
- 14.- HERNANDEZ FABELA AMALIEL  
S. C. T.  
LIDER DE PROYECTOS  
AV. MICHOACAN S/N  
COL. TEPALCATES  
DELEGACION IZTACALCO  
691-76-01
- 15.- HERNANDEZ OLIVA EDUARDO  
DIV. EST. DE POSGRADO F. I.  
AYUDANTE DE PROFESOR "A"  
CIUDAD UNIVERSITARIA
- ROSA VENUS No. 25  
COL. MIXCOAC  
DELEGACION ALVARO OBREGON  
01420 MEXICO, D.F.
- PASEO DE LAS ACACIAS No. 21  
COL. PASEOS DE TAXQUENA  
DELEGACION COYOACAN  
04250 MEXICO, D.F.  
670-24-97
- GPE. VICTORIA No. 25  
COL. SAN ANTONIO TECOMITLA  
DELEGACION MILPA ALTA  
12100 MEXICO, D.F.  
915847 01-56
- EDIFICIO 4 DEPTO. 101  
CONJUNTO SAN BUENAVENTURA  
TLALNEPANTLA EDO. DE MEXICO
- PACHUCA No. 131-406  
COL. CONDESA  
06140 MEXICO, D.F.  
286-28-31
- OTE 239 No. 40  
COL. AGRICOLA ORIENTAL  
DELEGACION IZTACALCO  
691-76-01
- FRANCISCO OLAGUIBEL 3-6  
COL. OBRERA  
DELEGACION CUAUHTEMOC  
06800 MEXICO, D.F.  
761-15-52



23.- QUINTANA TORRES FERNANDO  
S. C. T.

24.- QUINTO LARA CARLOS  
S. C. T.

ANALISTA PROGRAMADOR SIST. ESP. COMP.  
AV. POPOCATEPETL No. 506-B  
COL. XOCO  
DELEGACION BENITO JUAREZ  
03330 MEXICO, DF.  
688-98-53

RETORNO 25 No. 48  
COL. AVANTE  
DELEGACION COYOACAN  
04460 MEXICO, DF.  
549-08-06

25.- RANGEL GUTIERREZ RAYMUNDO HUGO  
FACULTAD DE INGENIERIA UNAM  
PROFESOR TIEMPO COMPLETO  
550-52-15 ext. 3746

CALZADA ACUEDUCTO 161 ED. B.  
DEPTO. 34  
COL. LA PIEDAD HUIPULCO  
DELEGACION TLALPAN  
14380 MEXICO, D.F.

26.- RIVERA RAMIREZ MA. CELINA  
SENEAM S. C. T.  
TECNICO ESP. SIST. COMPUTO  
BULEVARD AEROPUERTO  
535-77-00

EJE CENTRAL LAZARO CARDENAS No. 1167  
COL. VERTIZ NARVARTE  
DELEGACION BENITO JUAREZ  
539-41-73

27.- RODRIGUEZ BARRON CLEMENTE JUAN PABLO  
DIV. EST. POSGRADO F. I.  
AYTE. PROFESOR "A"  
CIUDAD UNIVERSITARIA

JOSE RODRIGUEZ GONZALEZ No. 9  
COL. CONTITUCION DE 1917  
09260 MEXICO, D.F.  
691-08-43

28.- RODRIGUEZ CHAVEZ JAQUELINA  
DIREC. GRAL. CONSTRUCCION Y OPERAC. H.  
JEFE OFINA. AUDITORIA APOYO ADMON.  
SAN ANTONIO ABAD No. 231  
COL. OBRERA  
DELEGACION CUAUTEMOC  
588-32-27

SINALOA No. 326  
FRAC. JACARANDAS  
54050 EDO. DE MEXICO  
397-45-68

29.- ROSALES VALDERRABANO EDMUNDO  
FAC. DE ING. DEPTO. COMPUTACION  
AYUDANTE DE PROFESOR  
CIUDAD UNIVERSITARIA  
550-52-15 ext. 2790

CDA. EZEQUIEL ORDONEZ No. 25  
COL. COPILCO  
DELEGACION COYOACAN  
04360 MEXICO, DF..

30.- RUBIO CAMACHO IVAN  
S. C.T.

31.- RUIZ BLANCAS JOSE DE JESUS  
ALTA TECNOLOGIA  
DIRECTOR GENERAL  
CALLE TOLEDO No. 120  
72550. PUEBLA, PUE.  
43-46-53

2a. PRIVADA DE LA SOLEDAD No. 17  
COL. NVA. ANTEQUERA  
43-46-53



101

102

103

104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200

201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300

301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400

401

402

403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500

501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600

601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700

32.- SANCHEZ GARCIA JUAN MANUEL  
S. C. T.  
TECNICO  
AV. DE LAS COMUNICACIONES  
COL. DEL MORAL  
DELEGACION IZTAPALAPA

RIVA PALACIOS No. 117  
COL. CENTRAL  
DELEGACION NETZAHUALCOYOTL

33.- SANCHEZ FUENTES ENRIQUE  
S. A. R. H.  
JEFE DE OFICINA  
INSURNETES No. 30-2o. PISO  
COL. JUAREZ  
DELEGACION CUAUHEMOC  
06000 MEXICO, D. F.  
591-18035

MINAS No. 82  
COL. ALVARO OBREGON

34.- SEPULVEDA DELGADO RICARDO  
D. G. C. O. H.  
JEFE DE LA UNIDAD DEPARTAMENTAL  
DE SISTEMAS DE INFORMACION  
SAN ANTONIO ABAD No. 231-1er. PISO  
COL. OBRERA  
DELEGACION CUAUHEMOC  
06800 MEXICO, D. F.  
588-32-27

HELIODORO VALLE No. 340  
COL. LORENZO BOTURINI  
DELEGACION VENUSTIANO CARRANZA  
15820 MEXICO, D.F.  
552-59-90

35.- SOBERON D. ROBERTO  
S. C. T.

36.- TEMPLOS CARBAJAL ALBERTO  
FACULTAD DE ING.  
AYUDANTE DE PROFESOR  
CIUDAD UNIVERSITARIA  
550-52-15 ext. 3750

OTE 180 No. 182  
COL. MOCTEZUMA  
DELEGACION VENUSTIANO CARRANZA  
15500 MEXICO, DF.  
762-48-72

37.- VARGAS GUERRERO XAVIER  
UNIVERSIDAD NAC. AUTONOMA DE MEXICO  
ANALISTA PROGRAMADOR  
DIV. EST. POSGRADO F. I.  
550-52-15 ext. 4860

VISION DE ANAHUAC No. 9-9  
COL. UNIDAD INDEPENDENCIA  
DELEGACION MAGDALENA CONTRERAS  
10100 MEXICO, D.F.  
593-30-26

38.- VAZQUEZ HUITRON DAVID  
S. C. T.