CURSO DE ADMINISTRACION DE PROYECTOS DE INFORMATICA

FECHA	TEMA	HORA	EXPOSITOR
13 FEBRERO	SISTEMA DE INFORMACION ENFOQUE SISTEMICO CONTROL DE CALIDAD	17-21	ING. BENITO ZYCHLINSKI ING. MARIO PALOMAR
14 "	PLANEACION DEL PROYECTO	"	ACT. RICARDO JEREZ
15 "	DEFINICION DE REQUERIMIENTOS DEL USUARIO	"	ING. MARIO PALOMAR
16 "	DIAGRAMAS DE WARMER/ORR	"	LIC. VICENTE LOPEZ
17 "	DISEÑO DE LA ARQUITECTURA DEL SISTEMA	"	ING. BENITO ZYCHLINSKI
20 "	"	"	ACT. RICARDO JEREZ
21 "	CONSTRUCCION DE PROGRMAS	"	ING. BENITO ZYCHLINSKI
22 "	"	"	ING. MARIO PALOMAR
23 "	LIBERACION DEL SISTEMA	"	LIC. VICENTE LOPEZ
24 "	MESA REDONDA Y CASOS PRACTICOS	"	TODOS

EVALUACION DEL PERSONAL DOCENTE

CURSO: ADMINISTRACION DE PROYECTOS EN INFORMATICA

FECHA: del 13 al 24 de Febrero de 1984

	CONFERENCISTA	DOMINIO DEL TEMA	EFICIENCIA EN EL USO DE AYUDAS AUDIOVISUALES	MANTENIMIENTO DEL INTERES. (COMUNICACION CON LOS ASISTENTES, AMENIDAD, FACILIDAD DE EXPRESION).	PUNTUALIDAD	
1.	Arq. Ricardo Jérez Díaz de León					
2.	Lic. Vicente López Trueba					
3.	Ing. Mario Palomar Alcibar					
4.	Ing. Benito Zychlinski Zychlinska					
5.						
6.						
7.						
8.						
9.						

ESCALA DE EVALUACION: 1 a 10

EVALUACION DE LA ENSEÑANZA

SU EVALUACION SINCERA NOS AYUDARA A MEJORAR LOS PROGRAMAS POSTERIORES QUE DISEÑAREMOS PARA USTED.

TEMA	ORGANIZACION Y DESARROLLO DEL TEMA	GRADO DE PROFUNDIDAD LOGRADO EN EL TEMA	GRADO DE ACTUALIZACION LOGRADO EN EL TEMA	UTILIDAD PRACTICA DEL TEMA	
Sistemas de Información					
El Enfoque Sistemico en el Desarrollo de Programas					
Planeación en el Ciclo de Vida de un Sistema					
Control de Calidad					
Especificación de Requerimientos de Usuario					
Diagramas Warnier-Orr					
Arquitectura del Sistema					
Programación Estructurada					
Planes de Prueba					
Operación y Mantenimiento					
ESCALA DE EVALUACION: 1 a 10					

EVALUACION DE LA ENSEÑANZA

SU EVALUACION SINCERA NOS AYUDARA A MEJORAR LOS PROGRAMAS POSTERIORES QUE DISEÑAREMOS PARA USTED.

TEMA		ORGANIZACION Y DESARROLLO DEL TEMA	GRADO DE PROFUNDIDAD LOGRADO EN EL TEMA	GRADO DE ACTUALIZACION LOGRADO EN EL TEMA	UTILIDAD PRACTICA DEL TEMA	
	Mesa Redonda y Casos Prácticos					
ESCALA DE EVALUACION: 1 a 10						

EVALUACION DEL CURSO

CONCEPTO		EVALUACION
1.	APLICACION INMEDIATA DE LOS CONCEPTOS EXPUESTOS	
2.	CLARIDAD CON QUE SE EXPUSIERON LOS TEMAS	
3.	GRADO DE ACTUALIZACION LOGRADO CON EL CURSO	
4.	CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
5.	CONTINUIDAD EN LOS TEMAS DEL CURSO	
6.	CALIDAD DE LAS NOTAS DEL CURSO	
7.	GRADO DE MOTIVACION LOGRADO CON EL CURSO	

<u>ESCALA DE EVALUACION DE 1 A 10</u>

1. ¿Qué le pareció el ambiente en la División de Educación Continua?

MUY AGRADABLE	AGRADABLE	DESAGRADABLE

2. Medio de comunicación por el que se enteró del curso:

PERIODICO EXCELSIOR ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA	PERIODICO NOVEDADES ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA	FOLLETO DEL CURSO

CARTEL MENSUAL	RADIO UNIVERSIDAD	COMUNICACION CARTA, TELEFONO, VERBAL, ETC.

REVISTAS TECNICAS	FOLLETO ANUAL	CARTELERA UNAM "LOS UNIVERSITARIOS HOY"	GACETA UNAM

3. Medio de transporte utilizado para venir al Palacio de Minería:

AUTOMOVIL PARTICULAR	METRO	OTRO MEDIO

4. ¿Qué cambios haría usted en el programa para tratar de perfeccionar el curso? ____

____

____

5. ¿Recomendaría el curso a otras personas?

SI	NO

6. ¿Qué cursos le gustaría que ofreciera la División de Educación Continua?

7. La coordinación académica fue:

EXCELENTE	BUENA	REGULAR	MALA

8. Si está interesado en tomar algún curso <u>intensivo</u> ¿Cuál es el horario - más conveniente para usted?

LUNFS A VIERNES DE 9 A 13 H. Y DE 14 A 18 H. (CON COMIDAS)	LUNES A VIERNES DE 17 A 21 H.	LUNES, MIERCOLES Y VIERNES DE 18 A 21 H.	MARTES Y JUEVES DE 18 A 21 H.

VIERNES DE 17 A 21 H. SABADOS DE 9 A 14 H.	VIERNES DE 17 A 21 H. SABADOS DE 9 A 13 Y DE 14 a 18 H.	O T R O

9. ¿Qué servicios adicionales desearía que tuviese la División de Educación Continua, para los asistentes?

10. Otras sugerencias:

DIRECTORIO DE PROFESORES DEL CURSO:
ADMINISTRACION DE PROYECTOS EN INFORMATICA
1984

1.- ACT. RICARDO JEREZ DIAZ DE LEON
Jefe del Depto. de Sistemas de Uso
Generalisado
Secretaría de Programación y Presupuesto
Izazaga No. 29 - 3°Piso
México, D. F.
Tel. 761 50 49

2.- LIC. VICENTE LOPEZ TRUEBA
Jefe del Departamento de Informática
Instituto Nacional para la
Educación de los Adultos
Subdirección de Planeación
Córdoba 23-4° piso
Col. Roma Norte
México, D.F.

3.- ING. MARIO PALOMAR ALCIBAR
Lider de Proyecto
S P P
Subdirección de Análisis e
Intercambio de Sistemas (SAIS)
Izazaga No. 29-3
Centro
México, D.F.
Tel. 761 50 53

4.- ING. BENITO ZYCHLINSKI (Coordinador)
Subdirector de Análisis e Intercambio
de Sistemas
S P P
Dirección de Política Informática
Izazaga No. 29-3
Centro
México, D.F.
Tel. 761 50 53 y 761 30 66

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

INDICE

ACT. RICARDO JÉREZ DÍAZ DE L.
LIC. VICENTE LÓPEZ TRUEBA
ING. MARIO PALOMAR ALCIBAR
ING. BENITO ZYCHLINSKI Z.

FEBRERO, 1984

ADMINISTRACION DE PROYECTOS EN INFORMATICA

REFERENCIAS

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

SISTEMAS DE INFORMACION
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

1

0

i. SISTEMAS DE INFORMACION
(DOCUMENTACION COMPLEMENTARIA)

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

3.1. Documento de Referencia nº 1

AUTOR: DIEBOLD, John	Clave: RF-
TITULO: INFORMATION RESOURCE MANAGEMENT -THE NEW CHALLENGE-	
IDIOMA: INGLES	TIPO DOC.: ARTICULO
REF.: Diebold Data Exchange Bulletin. IE/DS1/79V1-L1	

Information Resource Management —The New Challenge

by JOHN DIEBOLD

(This article comprises the Foreword to a special issue of the magazine Infosystems which commemorated the 25th Anniversary of the Diebold Group)

It is with great pride and pleasure that we in The Diebold Group have accepted the invitation to compile the articles for this special issue of Infosystems. We are thus able to celebrate the happy occasion of two anniversaries - the 20 for Infosystems and our own 25th.

When we consider the speed with which data processing technology has evolved and the enormous impact it has had upon business and management practices - and, in fact, upon many areas of life - we are aware that we have been part of a real revolution, a radical and pervasive change in the business environment and structure, and we are gratified by the role we have played.

High technology as a corporate resource has produced remarkable economic accomplishments. The business community has witnessed affordable computing go from the level of the enterprise in the 1950s, to the level of the division in the '60s, to the level of the department in the '70s, and in the 1980s, it will reach the ultimate affordable level of the employee. Along the way, it has lost its aura of an esoteric discipline and has become accepted as a practicable tool available to a broad community of users.

INFORMATICA

Both Infosystems and The Diebold Group have chosen to commemorate their anniversaries, not with a nostalgic look back over the last decades, but by delineating the challenges and opportunities that lie ahead.

It is useful, I think, to begin by providing an executive overview of the role of information management in the 1980s, a role that will change in many fundamental ways. Among the areas that will undergo alteration and that can b anticipated and planned for, are the information manager's span of control, the proliferation of computer hardware, and the expanded range of decision-making and business transactions within the guidelines of corporate information management.

Today, many corporations and institutions are quite satisfied with their data processing capabilities and so they should be; through much hard work these dat processing capabilities have become very effective. Nevertheless, the trap to which many executives in these companies may fall victim is the belief that information needs and requirements for information management during the 1980s can be satisfied by the principles and techniques that are proving successful today. It is assumed that the agent that will allow current directions to be maintained is technology itself, cheap minicomputers and new communications potential.

The calculator in the hand of an employee in the 1970s is likely to be a minicomputer in the 1980s. With this level of technological penetration, will MIS lose its ability to plan for and control the installations of computing devices and their information architecture? Can corporate goals be pursued ir this environment, or will the technology be counter-productive? One danger is that current good management practices in data processing may lead to chasing localised solutions. Such an orientation would lock corporations into a 1970s mode, a mode unresponsive to the business pressures of the 1980s. To be successful in utilising the new computer technology, it is necessary for executives to realise that the corporation needs a new planning orientation. Organisations and information managers should now be on a new learning curve f these emerging management principles. In our Group, we refer to this future orientation as Information Resource Management (IRM).

One of the basic changes in corporate management in the next decade will be the ability to handle information in more effective and efficient ways than we hav done in the past. I am not referring to data, but to information, which is the analysis and synthesis of data. In the role of information handling, I have always felt the safest comment I have ever read to be, "The modern age has a false sense of superiority because of the great mass of data at its disposal, but the valid criterion of distinction is rather the extent to which man knows how to form and master the material at his command". This passage was written by Johann Wolfgang von Goethe in 1810.

In the 1980s, one principle will characterise good information handling: the management of information as a corporate asset. In classical economic terms, the factors of production were viewed as land, labour, and capital. In the modern age, these have become men, money, machines, and material. In the future, I believe we will reach beyond this in defining the areas which deserve management concentration and include:

- work productivity and enrichment
- supply, as opposed to demand, management
- resource investment and conservation
- regulatory compliance
- consumer service
- product performance
- employee participation
- education
- information

As the goals of a corporation change with the complexity of the socio-political and competitive business environment, it becomes necessary to refine the structure of the areas on which executive attention is focused. Information will replace capital as the common denominator of corporate assets. Concerning the factors of production in the future, I feel that information will be recognised as a valuable resource comparable to capital and labour. We shall abandon this long period when - perhaps because of its abundance in a period of growing scarcities - information has been consistently underpriced, underutilised, and its contribution underrated.

It is clear that the organisations which will excel in the '80s will be those that recognise information as a major resource and structure it as efficiently as they do other assets.

- They will use information as a capability that allows them to change tactical plans more rapidly in order to have more cost-effective resources.
- They will use information primarily for planning and decision-making, rather than limiting it to the controls for which data is most widely used today.
- They will use information to measure performance and the organisation's responsiveness to its growing and varied constituencies, to complement the profitability measurements used today.
- They will use information channels to form instantaneous bonds with the consumer for marketing purposes: for tracking product performance, liability, and product maintenance on a longer-term basis.

Information which now loosely envelopes a product or service will become integral to it. The information package, electronically linked to the corporation, will be a vital component of the end product. Consumers will favour technical and service specifications over purely visual appeal or unsubstantial product claims.

As information-managers and architects within your organisations, you may well find that these changes will place you in a central role - if you grasp the opportunity. If, on the other hand, the hundred dollar minicomputer technolog is allowed to go unharnessed, indicating total freedom from information management, it is equally clear that your organisations will be at odds with demanding consumers or vendors; the impact will be visible on the P&L statemen as well as on the production line. For example, banks without customer terminals have experienced this same negative phenomenon within only a few years.

To manage the information needs of the 1980s, information architectures will have to be redesigned. It is important to emphasize that these architectures are not of the conventional data type prevalent today in data processing applications. Most of these applications are the result of formalising the handling of routine data within functional areas of the organisation. Compute applications have not, as a rule, dealt successfully with information that is i use between organisational functions or among the various constituencies of th enterprise. Data integration has long been an operational problem child, in th past, because we lack organisation and methodology to structure the data for integration. I contend that, in today's business, most of the clerical force i involved with unstructured information used in decision-making and transmitted by correspondence. It is organised only in so far as manual procedures exist and to the extent that local supervision enforces them.

The evidence for this assertion is practically indisputable, the clerical workforce is growing at a remarkable and very alarming rate. Recent U.S. Department of Labour statistics reveal that the clerical workforce surpasses, b close to five million people, that of either the operative (i.e. production workers) or the professional and technical workforce; it exceeds the number of service industry employees by an even greater amount. The growth rate of the clerical workforce has been matched only in the last five years by that of the professional and technical sector. Yet in Group studies with more than 100 major corporations, we have found that only 15% of these organisations were implementing office automation to improve productivity in the clerical area. 50% of these large corporations were not planning any automated measures whatsoever to deal with this issue.

If handling increased levels of data were the problem, we would have the solution in hand: our large computers and existing communications links. Corporate data is more easily managed than corporate information. However, even though the volume of data is increasing, it is the capture of corporate information with various degrees of "softness" that presents us with the challenge in the next decade. General management is now largely unaware of wha these information categories are; the information itself is buried at functiona department levels where often only one or two clerks know how it is synthesised from available data and operational level information exchange.

I would like to sketch briefly what the infrastructure for an industrial information activity is today. At the heart of the average operations area of a business - suppose it to be an operational department hotbed like production control, claims processing, or order entry - we can draw up a composite image. It is a functionally structured process of generating and acting on locally sourced information. The information is generated from computer lists, by paper business forms, by telephone consultations, by management decisions, and by general office procedures. It is significant that the sale of manifold business forms in the United States surpasses the worldwide shipment of minicomputers. Add to that fact the costs of telephone calls involved in fulfilling business transactions and the annual sales of filing cabinets, and the center of gravity of information technology today becomes painfully apparent. The information equipment industry and systems profession are still in their infancy; the 1980s promise to be their adolescence. The skill inherent in conducting a business transaction or reaching an operational decision in this manually based environment is directly related to operational knowledge born of experience. It is handed down from employee to employee within the department. When good and useful planning or control information is quickly generated in such a department, we evaluate the management of the department to be successful.

But this information infrastructure is vulnerable to several factors, for example, the accuracy of data, the reliability of manually created information, and work load fluctuations. Increases in the clerical workforce are frequently neeeded because there are no economies of scale in handling unstructured information within functional areas of the business.

That we know today what information will be needed in the next decade is a myth. It is foolhardy to believe that we can successfully predict the very information elements that the government, unions, and competitive pressures will demand. What organisations can do, however, is to plan how to formalise the handling of unstructured information, to design information infrastructures, to learn how to structure data in ways that will provide for an easier synthesis of strategic information planning and decision-making, and to find ways in which information - when viewed as a corporate asset - can be used to contribute additional value to other assets of the organisation.

Building a new information infrastructure requires a long lead time, and most of you - who are not yet doing so - should move in this direction soon. With that in mind, let us consider:

* The new needs of organisations.

* How our present information approaches are inadequate to meet these needs.

* Ways in which you, as information managers, can lead your companies in new directions.

The first three-quarters of the 20th century provided technological inventions that gave the major competitive advantages to large organisations. We moved from the assembly line and specialisation, requiring size and mass marketing, t the post-war sales and marketing boom, keeping capital intensive equipment full utilised, to R&D economies of scale.

In the last quarter of this century, we are seeing changes in the environment that, for the first time, do not make a large scale automatically preferable. Unless we move immediately, our relatively slow-moving, large organisations may be threatened. Some of the changes that we are beginning to see, and will see more of, are:

<u>Supply uncertainties</u>: "Demand management" is where we placed the emphasis in the past. This is now changing to emphasis on "supply management". Potential scarcities, balance of trade problems, nationalism in resource rich countries, and many other factors will introduce increasing uncertainties which will force organisations to frequently review their productivity. These factors will brin about marketing-purchasing-administrative adjustments that small organisations are ready to make.

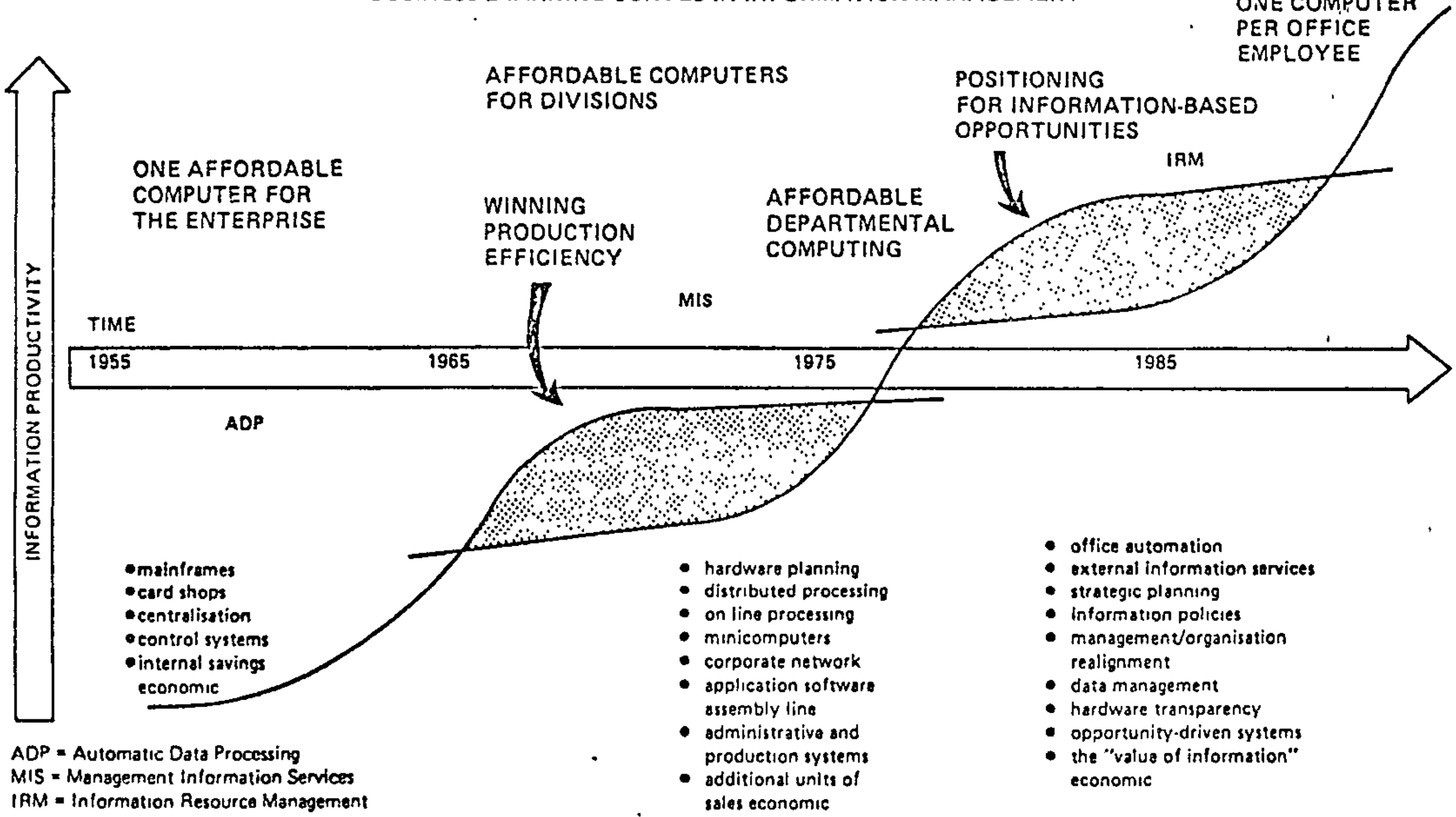
<u>International uncertainties</u>: The huge resource transfer to OPEC countries, among other factors, has already produced major instabilities that are likely t continue. They require fast response in order to adapt to them.

<u>Profit squeeze</u>: In an environment of rising costs and, sometimes, controlled price increases, organisations must constantly optimise their operations to remain viable. This, again, requires systems that will allow an optimisation of fast-moving variables, or a small enough organisation to accomplish it.

<u>Proliferation of areas in which the corporation is evaluated</u>: Social measurements of performance are being added to profit and loss measurements. This is another source of pressure on large organisations. The disappointment with large organisations and institutions, although commonly described as a communications problem, is undoubtedly at least partly related to a corporatio capacity to be more responsive to the increasing demands made on it by its various constituencies: personnel demanding more flexibility in their work hours and more responsibility in their work; customers requesting more individually tailored products than our traditional mass production can deliver; governments requiring faster response to new regulations; special interest groups insisting on better response to their social demands; foreign countries and communities requesting stronger commitments to their goals; and shareholders looking for profits increasingly difficult to achieve in a fast-changing environment.

It would be a serious corporate mistake to extrapolate the successful data processing management principles used now into the strategic directions for

BUSINESS LEARNING CURVES IN INFORMATION MANAGEMENT



INFORMATION PRODUCTIVITY
TIME
1955
1965
1975
1985
ADP
MIS
IRM
ONE AFFORDABLE COMPUTER FOR THE ENTERPRISE
AFFORDABLE COMPUTERS FOR DIVISIONS
WINNING PRODUCTION EFFICIENCY
AFFORDABLE DEPARTMENTAL COMPUTING
POSITIONING FOR INFORMATION-BASED OPPORTUNITIES
ONE COMPUTER PER OFFICE EMPLOYEE

- mainframes
- card shops
- centralisation
- control systems
- internal savings economic

- hardware planning
- distributed processing
- on line processing
- minicomputers
- corporate network
- application software assembly line
- administrative and production systems
- additional units of sales economic

- office automation
- external information services
- strategic planning
- information policies
- management/organisation realignment
- data management
- hardware transparency
- opportunity-driven systems
- the "value of information" economic

ADP = Automatic Data Processing
MIS = Management Information Services
IRM = Information Resource Management

α

corporate information processing in the future firm, as I have intimated earlier. Instead, I would like to outline five strategic coordinates for executives' efforts to set the right directions for information management in the 1980s.

First, executives should recognise that data processing technology is not information technology. Additional resources and methodologies are needed, for example, in the areas of office automation equipment, data base technology, small business computers, data capture equipment, and new communications links for voice, data, and image transmissions. These tools, in turn, must be managed to provide a new economy of scale in information management, if we want to make significant gains in productivity. These storage, transmitting, and processing devices can be at the information level rather than at the data storage level, which more than ever, should be tightly controlled for security, privacy, and backup reasons. They will be managed by policies rather than by direct authority.

Second, the natural candidate for managing the converged resources of data processing, communications, and office automation is the current Management Information System organisation and its chief executive. Discipline of business analysis, system economies, development and maintenance are needed to build the information architecture of the future. These acquired skills can easily be applied to the new, more global tasks. Information integration will come, I believe, through the convergence of resources management, not through the dispatch of a few systems analysts into a decentralised environment with the charge of building one or two strategic information networks among literally hundreds of locally owned computing devices.

Third, the span of control of the MIS executive will not be proportionally greater under an Information Resource Management orientation. It may well be reduced through the dispersion of computing equipment. However, replacing the reduction in the span of control will be a much broader concept of strategic planning and policy control. Resource management is not equatable with resource ownership in this context. With the programming onto computer chips of many business tasks that were, in the past, manual procedures, we can begin to see the emergence of information management policy as the logical replacement for these business procedures.

Fourth, there is a tremendous need for education and training in the area of information management. I fear that the cause of the shortfall of systems analysis and programming today are, in many ways, similar to those of operations research and management science at the turn of the last decade, although on a much larger scale. There is disenchantment with data processing careers. The root of the dissatisfaction, I believe, stems from the feeling of not being in the mainstream of the business. Information processing in the 1980s will cure this, but the task for executives now is to provide early motivation for a new breed of business analyst.

The fifth guideline concerns the challenge of building information architectures that will reach out to the customer, vendor, and other corporate correspondents. Using information to conserve resources, to secure greater consumer acceptance, to institute and fulfill compliance with the growing regulations of our complex society, to plan for greater profits, and to make our work efforts more productive in general, certainly must be major corporate objectives. Our current problems are not those of technology, but rather those of attaining management techniques and organisations at a higher level than that of data. Our profession has graduated from data processing to managing information for the enterprise. In a recent study, by our Group, of top U.S. companies, we found that more organisations consider themselves closer to the ADP environment of the 1960s than to the IRM environment of the 1980s. Despite this, practically all felt that they were technologically advanced. To me, this is a signal for some strong executive action.

It has been the preoccupation of information observers over the last two decades to monitor developments in information technology and assess their implications for the corporation. Today, we stand in the midst of a period of fundamental change in the corporate environment, and we face a challenge to the effectiveness, even to the survivability, of corporations. The distinguished contributors to this anniversary issue of INFOSYSTEMS will introduce ideas to ensure that corporations will not only survive, but prosper. Their articles develop new directions within the framework that I have suggested.

INFORMATICA

••• FORECASTING DEVELOPMENTS IN INFORMATION TECHNOLOGY FOR TWO DECADES

TIME FRAME	ORIENTATION OF ORGANISATIONS	THE DIEBOLD GROUP FORECASTS	FORECAST THE KEY ISSUE FOR:
1960s	• BATCH MIS HARDWARE • MIS INITIATED-APPLICATIONS • EFFICIENCY	• ON-LINE SOFTWARE • USER INITIATION OF APPLICATIONS • EFFECTIVENESS	1970s
EARLY 1970s	• ON-LINE CENTRALISED MIS WITH TERMINALS • LARGE COMPUTERS	• DISTRIBUTED PROCESSING • MINICOMPUTERS	LATE 1970s
MID 1970s	• STANDARDS	• INFORMATION POLICY	LATE 1970s
LATE 1970s	• DATA BASE ADMINISTRATION	• DATA MANAGEMENT	LATE 1970s
3 YEAR PLAN: 1979 To 1981	• MIS ORGANISATION (utility data processing)	• IRM ORGANISATION (office automation, communications and distributed data processin	EARLY 1980s
	• ACTIVE DATA DICTIONARIES	• INFORMATION INFRASTRUCTURES	MID 1980s
	• MIS EFFECTIVENESS	• VALUE OF INFORMATION	LATE 1980s

MIS = Management Information Systems; IRM = Information Resource Management

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

EL ENFOQUE SISTEMICO

DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

ii. EL ENFOQUE SISTEMICO

(DOCUMENTACION COMPLEMENTARIA)

2

2.0 EL PROCESO DE PROGRAMACION

Como señalamos en el capítulo anterior, en el costo de la programación (software) aumenta continuamente, mientras que el del equipo (hardware) disminuye año tras año. Además la mayoría de los problemas que se presentan en modernos y complejos sistemas digitales administrativos o científicos, son originados en la programación. Por esta razón es necesario proceder a su desarrollo en una forma sistemática y ordenada que se describe en este capítulo para llegar a producir programas que sean:

1. Utiles
2. Eficientes
3. Transferibles
4. Mantenibles
5. Confiables
6. Uniformes
7. Probables

En este capítulo se describe como la aplicación del enfoque sistémico permite desarrollar programación que cumpla con las características mencionadas.

Esta metodología prevee la división del proceso de desarrollo de programación en varias fases que siguen una secuencia temporal y se describen, incluyendo sus objetivos en los secciones de este capítulo. A saber:

Definición de Requerimientos

Diseño

Desarrollo

Operación y Mantenimiento

Cada fase a su vez se subdivide en etapas, que permiten, todavía más subidivir el problema de desarrollo para su realización y control.

Cada etapa en la evolución de la programación debe estar bien documentada, por lo que se mencionan especificamente los documentos que se generan en cada una de ellas.

2.1 Introducción: El Enfoque Sistémico

A medida que crecierón los requerimientos que se imponían a la programación aumentaron los problemas en su desarrollo, operación y mantenimiento: sobrecostos, retrasos en la entrega, insatisfacción del usuario, bajo confiabilidad y dificultad en el mantenimiento.

Siendo un programa un sistema complejo, no es de extrañarse que la problemática del desarrollo de programación se haya ido resolviendo, aplicando la metodología sistémica.

Si bien no existe una definición universalmente aceptada del término sistema, (todo autor que escribe sobre el tópico tiene la suya) podemos sin embargo afirmar que, una definición intuitiva adecuada sería la siguiente: un sistema es un agrupamiento de componentes cuyo comportamiento depende, tanto del de las partes, como de la forma en que éstas interaccionan entre sí.

Desde luego, resulta obvio que un programa complejo es un sistema, ya que el comportamiento del mismo no solamente depende de cómo trabaja cada una de sus partes, sino de la forma en que interaccionan entre sí. Por esta razón no es de extrañarse que los avances que se han obtenido en el desarrollo de programación estén basados en la aplicación de la metodología sistémica a este tipo de trabajos.

2.1.1 Dos Sistemas -

A continuación se citará la historia del desarrollo de dos sistemas, uno bien y otro mal planeado y se hará un paralelismo con el desarrollo de sistemas de programación.

La historia del desarrollo tecnológico de la humanidad está llena de implementaciones de sistemas cuyo efectos positivos han sido contrarrestados por sus efectos laterales negativos.

Dos ejemplos servirán para ilustrar la idea: uno planeado desde un principio tomando en cuenta su finalidad (de telefonía), y el otro que creció en forma no planeada (el de transportación individual por automóvil).

El sistema telefónico formado por teléfonos y una red integrada de componentes y equipos, ha hecho posible la comunicación inmediata entre dos puntos del sistema. Este sistema trabaja eficientemente y cumple con su finalidad, ya que desde un principio fue concebido como un sistema y no como un conglomerado de elementos independientes.

Supóngase por un momento, que el sistema telefónico hubiera evolucionado en una forma no planeada, sin tomar en cuenta los requerimientos sistemáticos. Algún industrial hubiera empezado a fabricar teléfonos a pequeña escala y los hubiera vendido a sus clientes. Estos hubieran contratado algún electricista para conectar entre sí los teléfonos de las personas con las que eventualmente hubiesen deseado comunicarse. A medida que el público se hubiera acostumbrado a hablar con amigos y otras personas, se hubieran requerido más y más hilos de teléfonos,limitando el número de hilos, la posibilidad de ofrecer servicio. Con este enfoque, si alguna persona deseara comunicarse con 1000 distintas localidades, de su teléfono deberían salir 999 hilos.

El municipio hubiera instalado algunos postes sobre los que el electricista hubiera colgado sus alambres. Pronto habrían habido demasiados alambres en los postes, y éstos hubieran empezado a caerse. El lector se reirá de este escenario, pero no difiere del de la transportación por automóvil, donde en las grandes ciudades las calles y vías rápidas ya están congestionadas.

Los requerimientos de nuestra sociedad, de contar con un sistema rápido, confiable y económico de comunicación, tarde o temprano habrían obligado a adoptar el enfoque sistémico en la planeación de este servicio. Afortunadamente para los usuarios del servicio telefónico, éste se desenvolvió desde un principio tomando en cuenta el enfoque sistémico.

No puede ser mayor el contraste entre el sistema descrito y el de transportación individual representado por el automóvil. Este sistema está formado por coches, carreteras, calles, refaccionarias, gasolineras, señales de tránsito, etc. Todo este conglomerado de subsistemas nunca fue planeado como un sistema integral, tomando en cuenta los objetivos del mismo: transportar personas eficientemente. Creció sin ninguna directriz!.

Como resultado de esta falla de previsión, el sistema, sobre todo en las grandes ciudades, ha llegado a su límite. Coches diseñados para correr a velocidades superiores a 100 kilómetros por hora, ruedan lentamente.

El sistema causa miles de accidentes, hace perder a sus usuarios miles de horas-hombre al día, contamina el aire que respiramos y es el culpable directo de más de un colapso nervioso. El sistema debió haber sido diseñado desde un principio para satisfacer las necesidades de transportación personal.

Mientras la demanda de servicio fue baja, esta evolución no planeada del sistema satisfizo los objetivos. Hoy en día, sobre todo en las grandes ciudades, este sistema dificulta gravemente la vida urbana.

Esta comparación entre la evolución de dos sistemas públicos ha ilustrado adecuadamente la importancia del enfoque sistémico en la planeación, implementación y operación de un sistema.

2.1.2 El Enfoque Sistémico Y La Programación. -

Originalmente el desarrollo de sistemas de programación no se iniciaba con una exhaustiva fase de planeación y definición de requerimientos. Se trataba de producir sistemas sin considerar que éstos, en un futuro, deberían de ser integrados en un sistema generalizado de información. Los problemas de integración hacían que el sistema, en el mejor de los casos, como el de transporte, trabajara ineficientemente. Se olvidaba que para poder planear adecuadamente un sistema de gran tamaño, es necesario emplear el enfoque sistémico, que es una evolución del método científico cuyas bases supo condensar tan bien el filósofo Dewey a principios del siglo. Este indicó que la solución de un problema, consiste en dar respuesta a las siguientes preguntas:

Cuál es el problema?.
Cuáles son las alternativas?.
Cuál es la mejor?.

El objetivo de la metodología de sistemas es precisamente desarrollar programación con la misma filosofía con la que se diseñó el sistema telefónico. El costo cada vez creciente de programación hace necesario seguir un método sistémico para su construcción y no uno de prueba y error, como venía realizándose hasta la fecha. La figura 2.1.1 muestra que el 55% de los errores en un programa son causados durante la fase de definición de requerimientos y el 50% de los errores aproximadamente son detectados durante el desarrollo.

La antigua tendencia general, desafortunadamente todavía no totalmente erradicada de iniciar lo más pronto posible la escritura de código, lleva a programas cuyos errores se detectan en fases muy adelantadas en el desarrollo del proyecto donde el costo de su corrección es muy elevada. (fig. 2.1.2).

```
FASES

               I
DEFINICION DE  IDDDD
REQUERIMIENTOS IGGGGGGGGGGGGGGGGGGGGGGG
               I
DISENO         IDDDDDDD
               IGGGGGGGGGGGGG
               I
DESARROLLO     IDDDDDDDDDDDDDDDDDDDDD
               IGGGGGGG
               I
OPERACION Y    IDDDDDDDDD
MANTENIMIENTO  IGGG
               -----+-----+-----+-----+-----+-----+------>
               0    10    20    30    40    50    60

                                                       % DE PROBLEMAS

                D - DETECCION
                G - GENERACION
```

FIGURA 2.1.1 GENERACION Y DETECCION DE ERRORES EN LAS DIVERSAS ETAPAS DEL PROCESO DE PROGRAMACION.

```
FASES
               I
               I
DEFINICION DE  ICCCCCCC
REQUERIMIENTOS I
               I
DISE~O         I       CCCCCCC
               I
DESARROLLO     I          CCCCCCCCCCCCCCCC
               I
OPERACION Y    I                     CCCCCCCCCCCCCC
MANTENIMIENTO  I
               ------+-----+-----+-----+---------+---------->
                     1     2     4     15        100

                                                       COSTO RELATIVO
                                                       DE CORRECCION
                                                       DE ERRORES
```

FIGURA 2.1.2 COSTO DE CORRECCION DE ERRORES EN LAS DIVERSAS ETAPAS DEL PROCESO DE PROGRAMACION

Por esta razón hoy se recomienda seguir el enfoque sistémico que identifica las siguientes fases para el desarrollo de programas computadora:

2.2 Definición de Requerimientos

2.3 Diseño

2.4 Desarrollo

2.5 Operación y Mantenimiento

A continuación se resumen los objetivos de cada una de estas fases y posteriormente se señalan las diferentes etapas que deben seguirse en cada una de estas fases. Debe señalarse que debe preceder al inicio de toda actividad de desarrollo de un sistema, sobretodo complejo, una fase de planeación, tema que se tratará en el capítulo 5.

2.2 Fase De Definición De Requerimientos.

No puede hablarse de un desarrollo sistémico de programas o sistemas de cómputo sin una precisa definición de objetivos y un claro planteamiento de requerimientos.

Los objetivos, deben definir las funciones generales que se esperan del producto final y en ellos se basa la estructuración del programa.

Los requerimientos definen con precisión las características de los programas de computadora por desarrollar y establecen los alcances del sistema de cómputo. Sirven de base para el diseño del sistema, de referencia para las pruebas del producto final y de fuente principal de información para el manual de usuario. Los requerimientos permiten además controlar la evolución del sistema durante sus etapas de desarrollo,

La fase de definición de requerimientos debe realizarse en 3 etapas:

1. Planteamiento de Objetivos

2. Análisis

3. Especificación de Requerimientos.

2.2.1 Planteamiento De Objetivos -

Esta etapa consiste en identificar y describir las necesidades del usuario, a fin de proponer un conjunto de objetivos, que de lograrse, implicarían la satisfacción de necesidades.

Puede aseverarse sin temor a equivocarse, que la claridad de los objetivos es condición necesaria para el éxito de todo proyecto. Una colección de objetivos imprecisos u obscuros puede propiciar una mala interpretación que eventualmente se traduzca en un producto final que no satisfaga las necesidades del usuario ó en un producto bastante más sofisticado de lo requerido para satisfacer sus necesidades.

Al establecer los objetivos deberá evitarse proponer o establecer un método específico como base para el desarrollo del sistema, a menos de que el empleo de dicha técnica en particular en constituya en sí uno de los objetivos del proyecto. La proposición de métodos específicos cae en el dominio de la siguiente etapa de Análisis. Estas actividades son responsabilidad del grupo de Análisis de Sistemas. (véase capítulo 5).

Se recomienda documentar formalmente el resultado de esta etapa en un "Documento de Objetivos del Proyecto" (DOP).

2.2.2 Análisis -

Al terminar la etapa de Planteamiento de Objetivos se aclaren las necesidades del usuario y los objetivos del proyecto, pero no se ha constatado aún la factibilidad de objetivos ni se han seleccionado, entre las alternativas viables, los métodos o medios a emplear para lograrlos.

La etapa de análisis consiste en estudiar los objetivos para establecer un compromiso entre los objetivos conflictivos ó antagónicos, establecer prioridades, constatar factibilidad y proponer los métodos de solución que servirán de base para el diseño del sistema.

Las actividades de esta etapa caen dentro del campo de acción de los analistas de sistemas. (vease capítulo 5). El grado de dificultad de esta etapa y el tiempo requerido para realizarla dependen directamente de la complejidad del proyecto o problema a resolver. El caso mas sencillo lo representan aquellos problemas que no requieren de aproximaciones para obtener su solución y el extremo más complejo lo representan aquellos problemas que, además de exigir algún tipo de aproximación o simulación, están relacionados con la mecanización de alguna actividad humana (LEH80). Algunas de las técnicas disponibles para realizar este análisis se presentan en la sección 3.2

Un análisis correcto de los objetivos es también una condición necesaria para el éxito del proyecto. Un análisis deficiente o incompleto de los objetivos puede redundar en el establecimiento de requerimientos y métodos que den origen a un producto incapaz de satisfacer los objetivos y las necesidades del usuario. En el caso de objetivos antagónicos, la indefinición de compromisos y prioridades puede dar lugar un producto final que solo satisfaga objetivos que el usuario considere como sofisticaciones irrelevantes y no los que, son importantes.

Como en el caso anterior, se recomienda documentar el resultado de esta etapa en un "Reporte de Análisis". (RA) Dicho reporte sirve para negociar y justificar ante el usuario, los compromisos y las prioridades adoptadas.

2.2.3 Especificación De Requerimientos -

Al terminar la etapa de Análisis se conocen las funciones del sistema y los métodos de solución que deberán ser empleados para realizarlas y satisfacer las necesidades del usuario. Esta información, reportada en los documentos DOP y RA, se redacta en un el lenguaje especializado, familiar a los analistas de sistemas (ingenieros electricistas, químicos, investigadores de operaciones, etc,) y no necesariamente claro e inteligible a los especialistas en computación (diseñadores y programadores) que eventualmente diseñarán y desarrollarán el sistema de computo. Los resultados de la etapa de especificación de requerimientos se resumen en un documento de especificaciones, que denominaremos 'Documento de Requerimientos de Programas de Computadora', (RPC), en el cual se traducen los resultados del Análisis a un idioma que permita la transferencia de ideas entre personal de distintas disciplinas. Este lenguaje está constituido por una combinación de prosa en lenguaje natural, con dibujos en algún lenguaje gráfico (diagramas de flujo de proceso, flujo de información, flujo de control, y notación matemática).

El RPC cubre además otra aspecto importante que hasta esta etapa no ha recibido atención: la interacción del sistema con el exterior: la llamada interfaz hombre-máquina y la interfaz con otros sistemas.

En síntesis, el RPC define lo que el sistema deberá ser capaz de realizar, establece la precisión con que deberán ser alcanzados los objetivos y describe el sistema de cómputo desde el punto de vista del usuario, en un lenguaje que sirve de medio de comunicación entre el usuario, el analista y los especialistas en computación.

El documento RPC deberá ser entendible, formal, completo y modificable (YEH 80):

* Entendible:El documento debe ser claro a las 3 partes involucradas:los usuarios, los analistas y los especialistas en computación.

* Formal: Las especificaciones deben redactarse de tal manera que todo requerimiento se identifique explicitamente como tal, no dando lugar a malas interpretaciones que redundarían en un producto que no satisfaría todos lo requerimientos ó que, al satisfacer aspectos que no son requerimientos, resulta más sofisticado de lo necesario.

* Completo: Debe cubrir todos aquellos aspectos que no deben dejarse al libre albedrío del diseñador. Todo aspecto que no sea explicitamente establecido en el RPC quedará libre para que el diseñador decida la forma de tratarlo.

* Modificable: El RPC debe ser estructurado, redactado y almacenado (procesador textos) de tal manera que admita cambios con un mínimo de esfuerzo y costo. El alcance del RPC termina donde comienza el dominio del diseño del sistema. Se deberá evitar el señalar la forma como el sistema de cómputo deberá ser diseñado para no restringir el campo de acción de los arquitectos del sistema. Determinar

la forma como el sistema será realizado es el propósito de la siguiente fase, donde se produce un diseño que obtiene el mejor provecho de los recursos de cómputo disponibles.

Una estrecha interacción con el cliente durante este fase de definición de requerimientos es la mejor garantía para lograr que la programación le sea útil (característica 1). Pero no solo permiten las acciones tomadas durante esta fase que cumpla el producto con esa primer catacterística. El empleo de algoritmos (procedimientos) adecuados analizado durante la etapa de analisis de esta fase, es uno de los principales factores que permite que la programación sea efeciente (característica 2). Un RPC, organizado en forma jerárquica y modular coadyuva a que la programación sea transferible (característica 3) y mantenible (característica 4).

Concluimos esta sección con un breve paréntesis para enfatizar la importancia de la fase de Definición de Requerimientos. Se ha mencionado ya en la introducción el alto porcentaje que representa el costo por mantenimiento sobre el costo total de un sistema de cómputo en su ciclo de vida (BOE73). Cabe señalar aquí como lo indican varios autores:((BELL 76), (BOE 76) y (PAR 78)), que gran parte de los costos por mantenimiento se deben a una especificación de requerimientos deficiente. El invertir suficientes recursos en el desarrollo de una fase de Definición de

Requerimientos sólida y completa es una acción altamente recomendable.

2.3 La Fase De Diseño

La computadora digital puede llevar a cabo actividades de procesamiento de información, una vez que le sean proporcionadas una serie de instrucciones u órdenes escritas en un lenguaje de programación. Los lenguajes de programación (Fortran, Cobol, etc) aunque presentan cierto parecido a los lenguajes naturales (español, inglés, etc), tienen una serie de restricciones con respecto a los últimos, y escribir instrucciones para una computadora es un proceso riguroso, formal y puede ser mas elaborado que la formulación de un problema usando notación matemática.

Los lenguajes de computadora son tan estrictos en su construcción que en lugar de afirmar que un programa esta escrito en un lenguaje se señala que esté escrito en código. Los lenguajes naturales son muy flexibles, expresivamente poderosos y por lo tanto pueden ser ambigüuos en el significado de sus construcciones. Por otra parte, los lenguajes de computadora o códigos no son tan flexibles y poderosos como los lenguajes naturales, pero si permiten eliminar totalmente las ambigüedades.

Por ejemplo, cuando se le dan instrucciones a una persona en lenguaje natural, en forma verbal o por escrito, es muy frecuente que la persona que las recibe tenga que aclarar las instrucciones a fin de confirmar si su interpretación es correcta o no. Esta situación no se presenta con las instrucciones escritas en código y ejecutadas por una computadora. La máquina toma una instrucción y la ejecuta inmediatamente; nunca existen ambigüedades ni conceptos que precisar.

Al proceso de transformar una serie de instrucciones en lenguaje natural a una serie de instrucciones en código se le denomina desarrollo de programas de computadora.

El RPC producto de la fase de Definición de Requerimientos, documenta las funciones que el programa de computadora debe de realizar, utilizando para ello una combinación de prosa en lenguaje natural, gráficos y notación matemática. Sus instrucciones todavía no puedan ser ejecutadas por una máquina. El RPC es, sin embargo, un paso muy importante en la transformación de una serie de instrucciones en lenguaje natural a una serie de instrucciones en código.

Sin el documento RPC que marca la terminación la fase de Requerimientos, no puede iniciarse la fase de Diseño. Esta será dividida en tres etapas:

1. Arquitectura.

2. Diagrama de estructura.

3. Detalle de módulos.

Estas etapas, y como todas las otras del proceso de programación, forman parte de un proceso iterativo. Al finalizar cada etapa, el producto es sometido a una revisión a fin de verificar y validar que satisface los lineamientos especificados en el RPC cumpliendo con las normas de control de calidad (capítulo 4) establecidas.

En la literatura, a las etapas del Diseño se les conoce con diferentes nombres: por ejemplo en [JET791], a la preparación de la arquitectura se le llama diseño del sistema; a la elaboración del diagrama de estructura se le conoce como diseño de la programación y al detalle de los módulos se le denomina diseño detallado.

2.3.1 Arquitectura -

Según el lineamiento fundamental del diseño de programas de arriba hacia abajo, los requerimientos tienen que dividirse en partes que resulten fáciles de entender; por lo tanto, es importante identificar las principales funciones que se encuentran implícitas en los requerimientos a fin de diseñar un programa de computadora para cada una de las funciones. La arquitectura de programas de computadoras precisamente identifica esas funciones y las asocia a un programa.

Asimismo, es indispensable definir durante esta etapa las interrelaciones externas a los programas de computadora, a fin de esclarecer las propiedades comunes a los programas, sean éstas físicas, funcionales o de procedimientos.

La documentación de esta etapa de Arquitectura (ARQ) deberá incluir la lista de los programas a desarrollar y las interfaces con cada uno de ellos.

2.3.2 El Diagrama De Estructura -

Una vez definidos el número de programas que se van a desarrollar y sus interfaces asociadas, se procede a elaborar un Diagrama de Estructura para cada programa, mismo que representa la estructura jerárquico y funcional de cada uno de ellos. En los sucesivo llamaremos a cada función con el nombre de módulo. Un diagrama de estructura debe documentar esta subdivisión en módulos y la relación entre ellos.

2.3.3 Detalle De Módulos -

En esta etapa del diseño, se detalla el proceso o función que representa cada uno de los módulos del diagrama de estructura mediante lógica estructurada. La especificación del proceso de cada módulo es descrita utilizando las técnicas de programación estructurada en un lenguaje de alto nivel denominado pseudo - código estructurado [LIN79], [ZYC81], que permite independiz ar el diseño del lenguaje final de cómputo empleado por el sistema de explotación. La documentación del Diagrama de Estructura y el Detallado de los Módulos se denomina Diseño del Programa de Computadora (DPC).

2.3.4 Selección Del Lenguaje De Programación. -

Con frecuencia, un lenguaje de alto nivel para una aplicación especifica es seleccionado en base a la experiencia individual restringida de una persona, en el conocimiento especifico de un idioma y en la inercia a aprender un nuevo lenguaje; es decir, la selección se hace realmente de una manera rápida y no cientifica. Muchos factores importantes no son tomados en cuenta al tomar la decisión.

Esta práctica debería abandonarse en favor de métodos más formales que tomaran en cuenta las ventajas y desventajas técnicas de los posibles lenguajes disponibles. Desde luego, habrá situaciones en donde sólo exista la posibilidad de seleccionar un lenguaje; pero en muchas otras la lista de alternativas es amplia. En estos casos hay que tomar en cuenta varios factores para la selección del lenguaje. La literaturá, en apoyo a la metodología de selección, es escasa. Sin embargo, existen algunas referencias interesantes. [AND82].

Anderson, Fujitsun y Shumate describen los factores a tomar en cuenta en la selección de un lenguaje, y aunque los citan para el caso de lenguajes para microprocesadores, la metodología es, sin embargo, aplicable a la selección de un lenguaje para un tipo de aplicación menos restringida. El factor más importante a considerar es la minimización del costo de vida de la programación, incluyendo, tanto el costo de desarrollo como el de mantenimiento, de acuerdo con la aplicación de la metodología de sistemas, citada alprincipio de este Capítulo. Los autores mencionados recomiendan formar un grupo de estudio con personas que representen un amplio rango de experiencia en campos relacionados con la selección de un lenguaje. Estos campos de experiencia podrían ser: Programación de Sistemas, Mantenimiento de Programación, Equipo, Administración de Programación de aplicación, entre otros. Investigación de Operaciones, Definición de requerimientos. La formación de un grupo evita, en la selección de un lenguaje, los sesgos individuales que pudieran existir si la selección la hiciera un solo individuo y además permite incluir una gama más amplia de características técnicas, como resultado de las recomendaciones dadas por el grupo.

[AND82] propone una metodología de tres etapas para la selección del lenguaje más apropiado:

- Inicial
- Intermedia
- De análisis formal

En cada una de las etapas antes mencionadas, se aplican criterios para eliminar posibles alternativas a ir reduciendo el número de opcionesa analizar en la etapa subsecuente.

Inicialmente,después de hacer una lista de los lenguajes disponibles ordenados por tipo: FORTRAN, PASCAL, PLI, etc., se procede a un análisis de tipo no cuantitativo para realizar la primer eliminación. Aquí se emplean criterios como disponibilidad y madurez del compilador y otros, como puede ser el que un lenguaje resulte francamente inadecuado para una aplicación específica.

Si ya se tiene seleccionado el procesador en que se va a operar, la no disponibilidad de un compilador para un lenguaje específico, será un criterio lógico para su eliminación

Para aplicaciones específicas, algunos lenguajes resultarán francamente inadecuados. Si la programación por desarrollar es de aplicación científica, considerar lenguajes como el COBOL, resultaría no adecuado.

Una vez que de la lista original de lenguajes se han eliminado aplicando los criterios anteriores, se procede a lo que podríamos llamar una etapa intermedia de selección donde se empiezan a emplear criterios cuantificables, como pueden ser:

a) Tiempos de ejecución de la programación.

b) Diversidad de proveedores.

Conviene eliminar, posiblemente, aquellos lenguajes donde sólo se tiene soporte de un proveedor o donde existan dudas sobre la continuidad de su presencia en el mercado.

c) Similitud entre lenguajes.

Si existen varios lenguajes similares, otro criterio sería eliminar aquellos que tienen menos apoyo por parte del vendedor o son más recientes y, por lo tanto, posiblemente todavía existan problemas con ellos.

d) Versatilidad

También deberán descartarse aquellos lenguajes que sean poco versátiles, que hayan sido desarrollados para un mercado en particular diferente a la aplicación para la cual se está considerando el idioma. Por ejemplo, para una aplicación comercial o científica, no es conveniente emplear un lenguaje que ha sido desarrollado para juegos.

En la siguiente etapa, que podríamos llamar de análisis formal, es necesario establecer ciertas figuras de mérito y peso para poder evaluar los lenguajes que hayan pasado por las dos etapas de selección anteriores (inicial e intermedia).

Durante la etapa de decisión formal se prueban características que idealmente deberían ser colectivamente exhaustivas, mutuamente exclusivas y no estar relacionadas. Desde luego, es imposible seleccionar un grupo que tenga exactamente estas características.

Entre los criterios mencionados en [AND 82] para la última etapa del proceso de selección pueden distinguirse criterios de tipo administrativos y de tipo técnico.

Entre los criterios de tipo administrativo, pueden destacarse los siguientes:

a) tiempo y costo de desarrollo

Estos deben de minimizarse pero, desde luego, no a expensas del incremento en los gastos de mantenimiento. Este criterio favorecerá características técnicas que facilitan la escritura de programación y minimizan los requerimientos de desarrollo de herramientas adicionales.

b) Mantenimiento durante el ciclo de la vida.

Este factor mide la facilidad con que un programa puede modificarse, con objeto de reducir sus costos durante su vida. Este criterio favorece características del lenguaje,como su legibilidad, y la riqueza y versatilidad de su estructura de datos.

Los factores técnicos pueden clasificarse como de:

- Primer orden
- Segundo orden

La clasificación anterior y su ordenamiento por rango de importancia depende, desde luego, de la aplicación específica que se tenga en mente.

a) Entre los factores de primer orden destacan:

- Representación de datos.

  Un lenguaje de programación debe ser juzgado por la riqueza en sus estructuras de datos, su posibilidad de realizar operaciones enteras, en punto flotante y complejos y tener estructuras de datos como listas ligadas, archivos, etc.

- Estructuras de control.

  Un lenguaje que tenga las estructuras de control básicas, como DO-WHILE, IF-THEN-ELSE, CASE, deberá preferirse a un lenguaje que no las contenga.

- Programación de sistemas.

  En el desarrollo y producción de programas relacionados con con el control y la operación de la computadora la supervisión, el mantenimiento, se requiere un lenguaje con la posibilidad de invertir, mover, y operar bytes entre otros.

b) Entre los factores de segundo orden se cuentan:

- Transportabilidad:

  Cuantifica la habilidad de un lenguaje para producir código reutilizable en diferentes procesadores.

- Facilidad de aprendizaje.

  Es un factor importante, sobre todo al tomar en cuenta la disponibilidad de recursos humanos para la escritura del código.

- Documentación.

  La calidad de la documentación de un lenguaje es un factor de alta importancia.

- Eficiencia en tiempo.

  Esta se mide en el tiempo de ejecución de programas 'tipo' [BENCH MARK]

- Eficiencia en espacio.

  Esta se mide por el tamaño del código objeto generado al compilar un programa 'tipo'.

- Facilidad de ligado con módulos de ensamblador.

  En algunos programas en tiempo real, es frecuente en usar lenguaje de ensamblador para algunas operaciones criticas en tiempo o porciones del programa que se usan con mucha frecuencia.

- Inteligibilidad.

  Es necesario que los programas puedan ser leídos y entendidos fácilmente para facilitar su mantenimiento y su actualización.

Estos criterios de tipo administrativo y técnico, deberán ser fijados por el grupo de estudio y empleando técnicas especiales, como por ejemplo, el método Delphos, se les pueden asignar pesos relativos a fin de establecer seguidamente (multiplicando el atributo con el peso respectivo), una figura de mérito para calificar los diferentes lenguajes de una decisión que afectará directamente la última de las actividades de la Fase de Diseño: el detallado de los modulos. Así como la Fase de Desarrollo del Proceso de Programación.

2.4 La Fase De Desarrollo

Los objetivos de la fase de desarrollo son:

- Codificación de los Módulos
- Verificación del Correcto Funcionamiento de Cada Módulo
- Integración de los Módulos para Formar Programas
- Integración de Programas para formar Sistemas.

La fase de desarrollo se considera terminada cuando el usuario acepta los programas de computadora integrados. Sin embargo, esta fase continua hasta que todas las modificaciones y discrepancias generadas por las pruebas hayan sido corregidas.

La fase de desarollo se divide en tres etapas:

1. Codificación
2. Integración
3. Pruebas de alto nivel

Los errores que se llegan a presentar en productos de programación pueden ser clasificados conforme al siguiente criterio:

- Error de codificación
- Error de análisis
- Error de especificación
- Error de diseño

Este criterio de errores es útil para describir los tipos de pruebas que se practican en ingeniería de programación. Los errores de Codificación son identificados durante la etapa de integración [2.4.2.]. Los restantes tipos de erros son tema de la sección denomiada Pruebas de Alto Nivel [2.4.3].

Probar un programa de computadora es el proceso de ejecutarlo con la intención de detectar errores. Núnca deberá procederse a validar programas de computadora bajo la suposición de que éstos están exentos de error.

La definición de las pruebas que habrán de llevarse a cabo es un arte, integrado por una serie de técnicas y recomendaciones basadas en la experiencia, que demandan ingenio y creatividad.

Las pruebas de un producto de cómputo se realizan mediante casos de prueba, definidos como un conjunto de datos de entrada y salida. Estos últimos datos son los resultados que produciría el programa de computadora al estar libre de errores si se le alimentasen los datos de entrada. Para las salidas de prueba se calculan primero los resultados mediante operaciones manuales o bien observando la respuesta en un sistema existente. Supóngase que un programa simula un sistema físico existente. A éste último se le aplican ciertas excitaciones (entradas) y se observan sus respuestas (salidas). Estos dos conjuntos (entradas-salidas) constituyen un caso de prueba. Un programa de computadora ante igual conjunto de entradas, si esta libre de errores, deberá producir iguales salidas.

En otros casos, mediante operaciones manuales, habrá que calcular las salidas correspondientes a las entrada para formar un caso de prueba.

La bondad de un caso de prueba deberá reflejar el potencial que éste tiene para detectar errores. Un buen caso de prueba es aquel que presenta una alta probabilidad de detectar fallas en la programación y un caso de prueba exitoso es aquel que efectivamente sirvió de base para corregir, al menos, un error hasta entonces desapercibido. En la sección 3.6 de este libro se discuten algunas técnicas de Diseño de Casos de Prueba.

En lo sucesivo se utilizará la palabra 'Prueba' para representar un conjunto de casos de prueba y al documento que describe las pruebas propuestas para un programa de computadora dado se le referirá como 'Plan de Pruebas' (PLP).

La ejecución de las pruebas requiere de una logística que considere aspectos tales como escenarios de prueba, equipo de cómputo, interfaz, personal, etc. El documento que repotará éstas condiciones se le denominará 'Procedimientos de Prueba'(PRP).

El establecimiento de una arquitectura compatible con el equipo de cómputo y plasmada en el documento de arquitectura (ARQ) y las acciones de Análisis llevadas a cabo en la fase anterior de 'Definición de Requerimientos' permiten desarrollar programas eficientes (característica 2). Documentación jerárquica y modular de : Arquitectura (ARQ) y de Diseño de Programas de Cómputo (DPC) que además debe tener un diseño estructurado son factores importantes para que la programación resulte transferible (característica 3) y (característica 4).

2.4.1 Codificación -

La etapa de Codificación de programas de computadora tiene como objetivo traducir las especificaciones de proceso de cada módulo, descritas en la fase anterior, en instrucciones ejecutables por un lenguaje de programación específico [FUJ 79]. La transformación de la definición del problema en especificaciones de proceso para cada módulo ha sido tratada en las secciones anteriores.

La programación o codificación de programas de computadora había sido considerada un arte hasta los años 50's, debido básicamente a la falta de métodos diseño de programas. Recientemente, y gracias al trabajo de gentes como Dijkstra, Wirth y Weinberg, se han empezado a desarrollar metodologías de programación basadas en el método humano de análisis y resolución de problemas.

Como se vió en la fase de diseño, un problema complejo no puede ser convertido en instrucciones de máquina o código de una manera natural y fácil. Es necesario convertir inicialmente la definición del problema en un lenguaje fácil de entender como el 'Pseudo-código', y posteriormente las operaciones descritas se refinan hasta que finalmente se escribe el código interpretable por máquinas computadoras.

Esta técnica es el corazón de varias metodologías de programación entre ella la 'Programación estructurada', la cual será descrita posteriormente en la sección (3.3) de Técnicas de Diseño.

El detalle del proceso a realizar, descrito en el documento DPC, marca la terminación de la fase de diseño y es un requisito indispensable para el inicio de esta etapa; ésta se considera terminada cuando todos los módulos han sido codificados y verificados; sin embargo las fases de diseño y desarrollo de programas pueden llevarse en paralelo dependiendo del enfoque que se tome.

Existen dos tipos de enfoques al respecto, uno radical y el otro conservador. El radical, se basa en el diseño de uno de los programas del sistema, procediéndose a su codificación y prueba inmediatamente. El enfoque conservador, por su parte, se basa terminar primero en terminar primero el diseño de todo el sistema, (integrado por diversos programas de computadora), pasando posteriormente a las etapas de codificación y pruebas para todo el sistema. Desde luego no existe una respuesta absoluta sobre cuál enfoque es mejor. Algunos factores que deben tomarse en cuenta para tomar una mejor decisión son:

- Conocimiento del usuario

Si el usuario no tiene idea bien clara de lo que quiere o cambia continuamente las especificaciones, use el enfoque radical; en cambio, si el usuario sabe con precisión lo que quiere, intente el diseño completo.

- Presiones de tiempo

Si se está bajo presiones de tiempo para producir resultados tangibles rápidamente, use el enfoque radical.

- Calendarios precisos

Si se requiere cumplir con calendarios precisos y detallados de utilización de recursos, debera optarse por el enfoque conservador. De otra manera, con el enfoque radical sería muy difícil determinar con precisión el tiempo de duración y los costos del sistema sin ni siquiera conoce cuantos módulos va a contener el programa.

Independientemente del enfoque elegido, los módulos de un programa deben ser codificados. Existen dos estrategías principales para codificar estos módulos:

- Codificación descendente (TOP-DOWN PROGRAMMING)

- Codificación ascendente (BOTTOM-UP PROGRAMMING)

Ambas estrategías son correctas, sin embargo la codificación ascendente permite atacar los módulos de los niveles jerárquicos más bajos primero. Estos contienen las operaciones primitivas, y permiten de ahí construir las operaciones más complejas. Una segunda ventaja de utilizar la codificación ascendente es que la integración de los módulos y los casos de prueba son diseñados y realizados siguiendo la misma secuencia como lo muestran las siguiente. secciones.

Todos aquellos módulos que han sido probados pasan a formar parte de la 'Biblioteca de Soporte de Programas', (BSP) una herramienta automática que será descrita en la sección 4.3 de este libro. Esta biblioteca tiene como función el almacenamiento de todos aquellos módulos que hayan sido aprobados, y que cumplan con los requerimientos del usuario establecidos en el RPC y verificados por control de calidad de acuerdo a las normas (capítulo 4) establecidas. Estos módulos, de ahora en adelante, se definirán como 'módulos reusables'. La función de verificación por control de calidad se conoce con el nombre de 'Auditoría' y se puede realizar con una segunda herramienta automática conocida como 'Auditor de código'. En la sección 4.3 se describe la función de un auditor de código estático.

Los reusables no son los únicos productos finales de esta etapa. Además de estos módulos de código fuente insertados en la BSP existen listados de impresora, módulos objeto, procedimientos catalogados, resultados de las pruebas y finalmente, y en caso necesario, las versiones actualizadas de los documentos RPC, DPC y Planes de Prueba.

Resumiendo, la etapa de codificación no deberá de iniciarse antes de completar la Revisión del Diseño del programa a codificar. Esto, sin embargo, no significa que se necesite tener todo el sistema diseñado y revisado, para empezar la codificación. Las normas de codificación u las herramientas automáticas para el control de esta programación son indispensanbles para esta etapa.

Los productos de esta etapa del proceso de programación quedan plasmadas en la documentación que se obtiene del código fuente generado (COD).

2.4.2 Integración -

El objetivo principal de esta etapa es la integración funcional de los módulos de un programa de cómputadora ajustandolos a las particularidades del sistema de explotación de cómputo.

En esta etapa de Integración deben tomarse en cuenta dos aspectos importantes: la manera como se combinan los módulos para formar programas y el diseño de pruebas que permiten identificar errores de codificación.

Pueden seguirse dos procedimientos:

Integración no incremental - Validación de programas de computadora a partir de pruebas modulares independientes.

Integración incremental - Validación de nuevos módulos (no probados)agregándolos a módulos ya probados e integrados.

Sobre estos procedimientos puede comentarse:

El proceso de integración no incremental requiere de un mayor esfuerzo. Consideremos, para fines ilustrativos, el diagrama de estructura del programa de computadora de la figura 2.4.1.

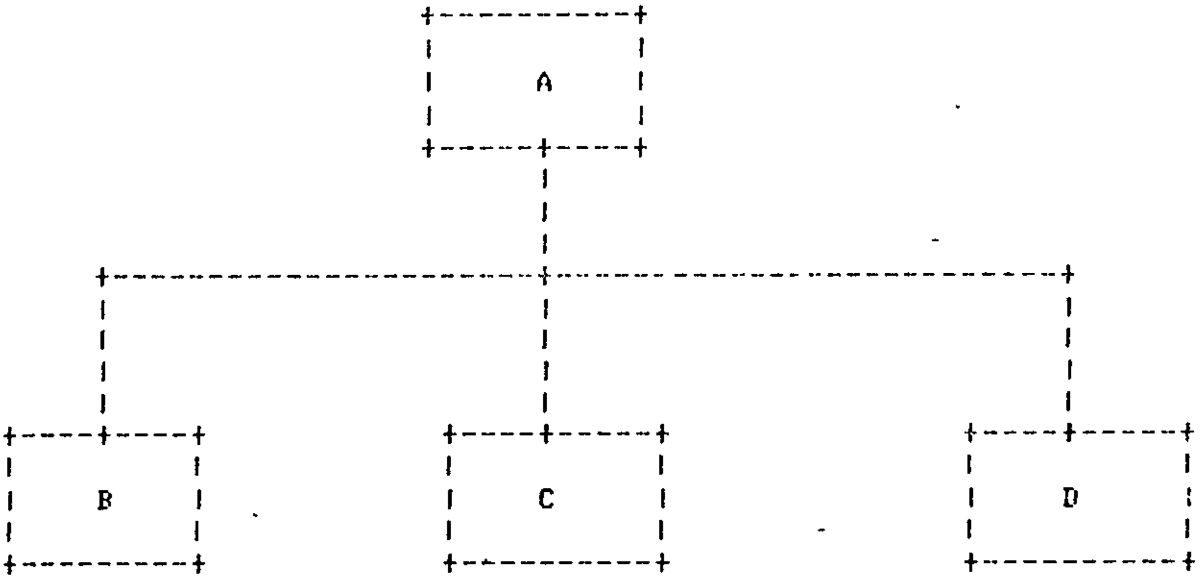


Fig. 2.4.1.
ESTRUCTURA DE UN SISTEMA EN PROCESO DE INTEGRACION

Por ejemplo, bajo la variante de integración no incremental, para probar el módulo 'B' es necesario diseñar casos de prueba y alimentarlos a través de un módulo 'direccionador', que a su vez despliegue los resultados obtenidos en 'B' Para probar el módulo 'A' deberán existir 3 módulos 'ciegos' que simulen las funciones de 'B', 'C', y 'D'.

Al utilizar la técnica incremental es necesario un menor esfuerzo, ya que por ejemplo, una vez probados individualmente los módulos "B", "C" y "D", podrán ser utilizados en la validación del módulo principal "A", evitando así el trabajo de elaborar los módulos "ciegos" "B", "C", y "D".

Integracion incremental permite detectar antes los errores de programación relacionados con la intercomunicación entre los módulos. Al integrar un nuevo módulo ya probado individualmente a un conjunto que ya también fue validado, un error podrá atribuirse en primera instancia a la interconexión. El esquema de integración no incremental no propicia esta coordinación sino hasta el término del proceso mismo.

Integración incremental permite identificar errores en los módulos del programa de computadora adicionados mas recientemente. Integración no incremental elimina esta identificación, ya que los errores en este caso, emergerían hasta que todos los módulos hubierón sido ensamblados; estos errores podrían estar en cualquier módulo del programa de computadora volviéndose muy difícil la identificación de los componentes incorrectos.

Dadas las tendencias en los sistemas de cómputo (costos decrecientes del equipo y crecientes en la programación) los errores de programación contribuyen a engrosar los costos de programación; detectarlos oportunamente coadyuva para disminuir el precio de la programación. La variante no incremental favorece precisamente esta detección oportuna de errores, por lo cual sugerimos preferentemente su empleo.

Es pertinente mencionar que el uso de herramientas automáticas puede simplificar el proceso de integración al eliminar, por ejemplo la necesidad de construir módulos 'direccionadores' (MYE 79). Existen herramientas que analizan el flujo de un programa a fin de encontrar instrucciones que nunca podrían ser ejecutadas y encuentran en que una variable es utilizada antes de serle asignado un valor. En la sección 4.3 de este libro se abunda sobre el tema [HAR 82]

2.4.3 Pruebas De Alto Nivel -

Después de la fase de integración en la que se han identificado los erroes de codificación, quedan por detectarse los de análisis, especificación y diseño.

Bajo el nombre de errores de análisis se conocen fallas no detectadas en la etapa de Análisis de la fase de Definición de Requerimientos, que impiden satisfacer las necesidades del usuario.

Errores de especificación, causados por especificaciones incompletas o imprecisas dan lugar a una interpretación equivocada de las mismas.

Un diseño imposible de implementarse en el ambiente del equipo de explotación de cómputo que se pretende utilizar, o incapaz de satisfacer los requerimientos del programa es la causa de los denominados errores de diseño.

El número de errores que se llegan a presentar en la etapa de integración puede ser minimizado si se revisa el producto al final o durante el desarrollo de cada etapa en la evolución del programa. Sin embargo la experiencia ha demostrado, que si bien las revisiones reducen drásticamente el número de errores, son insuficientes para identificar todos los problemas. Es por esta razón que se hace necesario someter el código a pruebas específicas, de alto nivel a fin de identificar el mayor número de errores que hayan pasado desapercibidos durante el desarrollo de la programación.

Se denominan "Pruebas de alto nivel", a aquellas que tienen como objetivo identificar, no los errores de codificación ya encontrados en la etapa de integración, sino los de análisis, especificación y diseño.

Entre estas pruebas las más importantes son las siguientes:

- Pruebas funcionales
- Pruebas de implantación
- Pruebas de sistema
- Pruebas de aceptación

2.4.3.1 Pruebas Funcionales -

El objetivo de estas pruebas es encontrar errores de análisis y de especificación cometidos en las etapas de análisis y especificación de requerimientos.

Los casos para este tipo de prueba son generalmente producidos mediante técnicas de análisis de entrada-salida (sección 3.6). Para este tipo de técnicas la lógica interna del programa es irrelevante.

Al realizar una prueba funcional, los documentos "RA" y "RPC" son analizados para que apartir de estos se diseñen y seleccionen los casos de prueba.

La busqueda de errores de concepto y de especificcción debe ser organizada. Para ello deberán diseñarse las pruebas funcionales jerarquizando los componentes a probar en distintos niveles de abstracción. Será conveniente por ejemplo, diseñar casos de pruebas para partes de dimensiones fáciles de denominar, tales como: tareas, funciones y aspectos interfaz hombre-máquina.

Durante esta etapa se producen, basada en las normas descritas en 4.2.4 y 4.2.4, los documentos de planes y procedimientos de prueba (PLP) y (PRP).

2.4.3.2 Pruebas De Implantación -

Generalmente los programas de cómputo son sometidos a pruebas modulares de construcción y funcionales en un sistema distinto al que serán finalmente integrados.

Las pruebas de integración tienen como objetivo encontrar errores de especificación, concepto y diseño en el ambiente real (equipo, sistema operativo, interfaces, etc.) donde los programas serán finalmente integrados.

Una práctica recomendada es volver a utilizar las pruebas funcionales para este propósito.

2.4.3.3 Pruebas De Sistema -

Esta prueba no consiste en volver a probar todas las funciones o programas de un sistema, proceso que se llevó a cabo durante la etapa de pruebas funcionales.

La fuente de información que sirve de base para el diseño de estas pruebas es el documento original de objetivos en que se basó el desarrollo del sistema de cómputo (DOP).

La revisión exhaustiva del 'Manual de Usuario' (MUS) es parte integral del proceso de prueba del sistema, y es la principal razón para realizar la prueba. El beneficio más importante (MUS) de esta prueba es la revisión de la documentación de usuario a fin de corregirla en caso necesario. Este manual que se elaboró en paralelo con la realización de estas pruebas de sistema es la documentación de caracter operativo, que permite al usuario de la programación su correcta exploración.

Cabe aclarar que la prueba de sistema no solo es relevante para sistemas de programas de cómputo sino que también es aplicable al caso de programas aislados. El primer contacto entre muchos usuarios y los programas son los manuales. Un buen manual, fácilmente entendible y sin erroes facilita, no solo el empleo del programa por parte del usuario, sino a veces, inclusive, su aceptación por gentes escépticas al empleo de estos medios de manejo de información.

2.4.3.4 Pruebas De Aceptación -

Las pruebas de aceptación tienen como objetivo comparar el producto de cómputo final con el contrato original.

Generalmente es el usuario el responsable de esta prueba. En el caso de sistemas grandes y complejos, es práctica recomendada delegar la responsabilidad de esta prueba a una tercera institución, distinta del usuario y de la organización que desarrolla el producto. Tanto durante la fase anterior de 'Diseño' con la presente de 'Desarrollo' se llevan a cabo acciones diversas que garantizan que la programación sea probable (característica 7)

2.5 Fase De Operación Y Mantenimiento

Esta última de las fases del proceso de programación es donde se reflejan los aciertos o los errores de las fases previas. Se identificarán aciertos en la medida en la que los requerimientos de los programas de computadora satisfacen las necesidades del usuario, la arquitectura y los diseños se asocien a las características específicas del sistema de explotación de cómputo y en general, la disciplina que hubiere sido empleada para la construcción del código sea formal. Asimismo, gran parte del éxito de la programación radica en la metodología que se emplee para controlar los cambios que se tengan que realizar a los programas después de la primera instalación; controlar las diferentes versiones del código redunda en un beneficio inmediato para el usuario.

Por otra parte, es ésta la fase en donde suelen escucharse frases como la siguiente:

- El programa trabaja muy bien, pero esto no es lo que yo deseaba!!.

- Los programas aislados funcionan perfectamente, pero el sistema de programación no trabaja!!.

- El código que nos fue entregado inicialmente funcionaba; ahora desconocemos que ha sucedido, pero ya no funciona!!.

Todas estas observaciones señalan que la metodología utilizada en el desarrollo de esa programación fue deficiente: iniciando con los requerimientos, estos nos satisfacen las necesidades del usuario; el no establecimiento de una documentación de interfaz entre los diferentes programas del sistema de cómputo originaron desajustes que evitan el éxito en la integración de la programación; una documentación deficiente y una administración no apropiada de la programación obligan a invertir grandes cantidades de recursos humanos para restaurar la programación.

La fase de operación de la programación se inicia con la primera instalación del sistema de programación integrado, una vez que la programación ha sido aceptada por el cliente en base a los documentos de los Planes de Prueba (PLP) y los Procedimientos de Prueba, (PRP) descrito en el capítulo 4.2.4. y 4.2.5. La documentación del usuario, normalmente llamada Manual de Usuario (MUS) permite a los operarios de la programación utilizarla cabal y eficientemente según las especificaciones del usuario vertidas en el documento de requerimiento de la programación [RPC]. A partir de este momento en adelante, se inicia un proceso de aprendizaje por parte de los operarios de la programación. Es así como la creatividad, el ingenio y los deseos de superación del operario obligan al mantenimiento de código.

2.5.1 Mantenimiento -

Dentro del campo de programación es necesario aclarar el término mantenimiento, que es generalmente utilizado para describir todos aquellos cambios hechos a la programación después de su primera instalación. Por ello difiere significativamente de su concepto general, que describe la restauración de un sistema o componente a su estado original. El deterioro que ha ocurrido como un resultado del uso o del paso del tiempo es corregido a través de una reparación o de una sustitución. Programación no se deteriora espontáneamente por una interacción con el medio. La programación no cambia a menos de que el personal la haga cambiar, como resultado de un deseo del usuario de eliminar errores, o de un funcionamiento inapropiado o restringido.

En equipo, término utilizado para hacer referencia a las componente físicas de un sistema, cambios mayores a un producto son logrados a través de un rediseño o a través de una nueva construcción del modelo. En programación, mejoras y adaptaciones son consumadas a través de eliminaciones o extensiones al código existente. Nuevas características comúnmente no señaladas durante las fases iniciales del proyecto, se imponen a la programación original sin un rediseño total del sistema.

En un documento reciente, Lientz y Swanson (LIE 82), mostraron que para sistemas de procesamiento de datos administrativos los costos de mantenimiento exceden a los costos de desarrollo. De acuerdo con Swanson, los recursos para mentener programación son generalmente aplicados a tres cláses de esfuerzos:

a) Corrección de problemas latentes

b) Perfeccionamiento de programas y mejoras en la documentación

c) Cambios adaptivos al medio ambiente de la programación.

Lientz y Swanson distribuyen los esfuerzos de mantenimiento aproximadamente en la siguiente forma:

1) Corrección de problemas	22%
2) Reformas a programación y documentación	51%
3) Cambios y adaptaciones	27%

A continuación se presentan algunas acciones que pueden ser implementadas en las fases de Definición de Requerimientos y de Desarrollo y que redundarán finalmente en la eliminación parcial o total de algunos de los principales problemas que se presentan en el mantenimiento de programación.

2.5.1.1 Acciones Correctivas -

1. Probar cada instrucción y cada rama de programación durante la integración de los módulos (sección 3,4.2.).

2. Rastrear y verificar la demostración de todos los requerimientos en alguna etapa de las pruebas. Esto puede lograrse mediante una "Matriz de Requerimientos", (4.1.4) que válida dentro de los Planes de Prueba cada

uno de los requerimientos señalados en el documento RPC.

2.5.1.2 Acciones Perfectivas -

1. Muchos problemas ocurren debido a que la especificación de los requerimientos fue incompleta o poco clara; es necesario buscar potenciales omisiones en la especificación.

2. Proponer al usuario, tan pronto sea posible dentro del proceso de programación, las facilidades de interfaz hombre-máquina que tendrá en el sistema; comúnmente son necesarias mejoras en esta área.

2.5.1.3 Acciones Adaptivas -

1. Es indispensable desarrollar código mantenible que utilice normas de documentación y codificación estructurada.

2. Desarrollar código módular, simplificando interfaces y limitando el número de instrucciones ejecutables en cada subrutina.

3. Actualizar la documentación, no entregando nuevas versiones de código sino hasta que toda la documentación esté completa.

4. Mantener el mismo nivel de control en la base de datos y en el código; utilizar control automático y centralizado cuando esto sea posible.

2.6 Resumen

En este capítulo se ha expuesto cómo el enfoque sistémico permite desarrollar programación compleja que cumple con los requerimientos del usuario. La metodología implica subdividir el proceso en bases y etapas, bien documentadas que se resumen en la tabla 2.6.1.

Los párrafos iniciales resaltaron los altos costos de desarrollo y mantenimiento de la programación que el enfoque expuesto permite minimizar. Aún lográndolas abatir, éstos seguirán siendo altos, pero los beneficios económicos de las aplicaciones potenciales de las computadoras son tan altos, que estos niveles de costo son en muchos casos perfectamente aceptables de programación. Es de esperarse sin embargo, que la investigación en el campo de desarrollo de programación (1.3.2.) siga no solo disminuyendo el costo relativo de su construcción, sino también permita que cada vez satisfaga en mayor grado las características enunciadas al principio de este capítulo. A medida que la humanidad depende más y más de la programación que controla las computadoras y que como consecuencia controla la sociedad, se vuelve indispensable que los usuarios controlen absolutamente la programación y los procesos a través de los cuales se genera el código. Para logar esto se requiere toda una teoría, modelos, metodologías, técnicas, herramientas; en fin una disciplina, vertida en el concepto de la ingeniería del proceso de programación.

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

CONTROL DE CALIDAD
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

iii. CONTROL DE CALIDAD
(DOCUMENTACION COMPLEMENTARIA)

INFORMATICA II - 7.4. - 6

STANDARES

AUTOR:	RICHMAN	CLAVE:	RF-1
TITULO:	THE STANDARS MANUAL		
IDIOMA:	INGLES	TIPO DOC.:	INFORME
REF.:	AUERBACH STANDARS AND PRACTICES		

The Standards Manual

PAYOFF IDEA *One of the major weaknesses of data processing as a profession, art form, or business is its lack of consistency and clarity This is true from the viewpoint of users managers, and the technical "professionals" who write programs, design systems, and generally "practice" data processing This problem has recently led to attempts to have data processors licensed by governmental agencies in order to impose order on a patchwork of confusing practices which have been construed as dangerous to many organizations and even to society The purpose of this portfolio is to point out another path, one which is both easier to follow and more attainable within your organization consistent standards for all data processing activities*

PROBLEMS ADDRESSED

Instead of tolerating the vague intuitions and bad training which data processing people often display in the development of a computer system, your organization can quickly and painlessly impose order, and then obtain many significant benefits by creating a concise standards manual

Characteristics of a Well-Run Data Processing Shop

Fundamentally, a successful DP organization is *well managed* Externally, users are satisfied because they are getting useful complete, accurate information which enables them to do their jobs effectively Their requests for changes additions, and deletions are handled promptly and accurately When problems occur there are good controls to quickly identify the problem isolate it and apply corrective measures Senior management sees order instead of chaos and it gets results from monies allocated instead of requests for greater allocations without clear (to them) justification Outsiders such as customers and government agencies see information being handled responsibly with beneficial effects to them as individuals and to society as a whole

Internally, systems and programming personnel have clear guidelines as to what is expected of them and how to proceed in the transactions which comprise their jobs. Computer operators have clearly defined responsibilities and totally clean, usable documentation to quickly identify and correct defects due to hardware malfunctions, software bugs, user-generated data errors, or operator errors.

All audit trails are clearly defined and are documented neatly and logically. Every program is consistently coded, documented, tested and protected through systematic security back-up procedures despite the variety of programmers, systems analysts, and project managers who have participated in their development. The effects of varied employee backgrounds, training, attitudes and aptitudes are negligible as work is productively funneled through a process which causes the results to meet a consistent set of *standards* for all activities.

How is this process accomplished? There are several ways. One is to have a very tough, hard-driving management team which will not settle for anything-less. Another is to have a data processing staff with homogeneous skills, attitudes and backgrounds plus a strong will to achieve the results just described. A third, and most realistic way is to have a management tool which causes diverse people to perform consistently under changing conditions. Such a tool is a good standards manual.

Need for Standards in Systems and Programming

Recognized procedures and standards are fundamental to the smooth controlled operation of a data processing organization. The reason for this is that *data processing*, when it is managed properly, *is a very precise set of activities* that results in a rather exact response to the user's requirements. Unfortunately, *people communicate imprecisely*, so the resulting systems do not always come close to what the user thought he requested.

The best illustration of this is Figure 1, the set of cartoons describing "The New System", which shows a variety of approaches to producing a swing tied to a tree. The fundamental message is that the swing was defined imprecisely by the user, then other parties, who did not understand the use of the swing perpetuated the vague qualities into designs which were useless or very impractical. In the real world, where users ask for systems which are more difficult to describe than swings, the lack of clarity can become even greater by the time the system has been designed and implemented. Moreover, the resulting frustration and anger which *all* parties feel can be enormous. The purpose of this portfolio is to define a methodology that would minimize gross errors in communication and lessen the probabilities of "kluge" solutions to vital systems problems.

TYPICAL SYSTEMS AND PROGRAMMING STANDARDS

There are four major areas which require detailed standardized specifications in order for systems to be reliably developed.

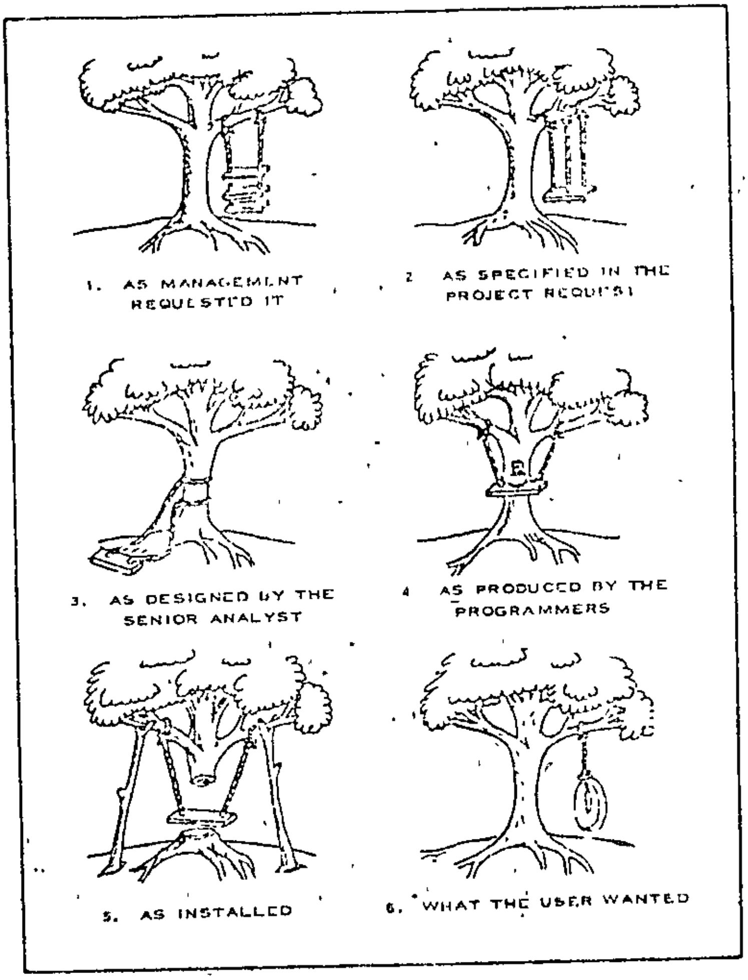
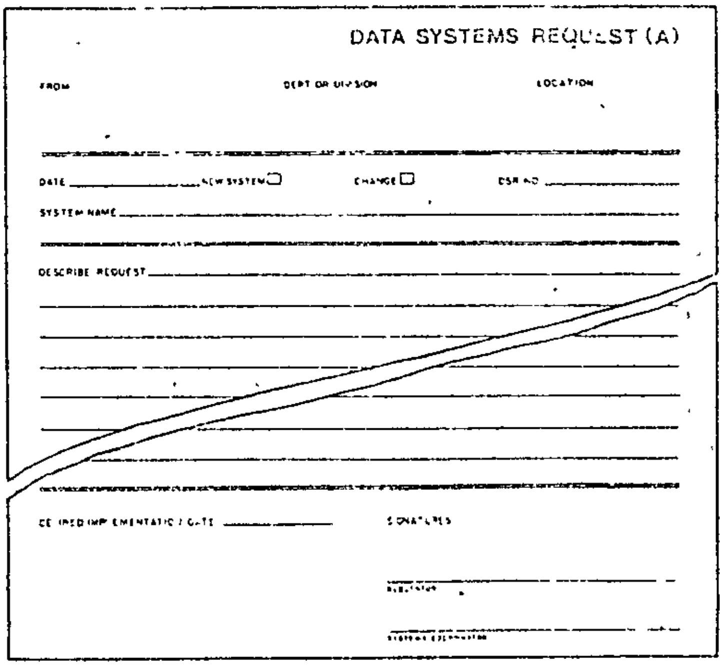

1. AS MANAGEMENT REQUESTED IT
2 AS SPECIFIED IN THE PROJECT REQUEST
3. AS DESIGNED BY THE SENIOR ANALYST
4 AS PRODUCED BY THE PROGRAMMERS
5. AS INSTALLED
6. WHAT THE USER WANTED

Figure 1. The New System

System Requirements

These needs must be thoroughly documented by the user and the systems analyst. The standards defined in this portion of the manual must provide for three major activities. First, the user's statements about his requirements must be thoroughly and formally defined. Second the systems department's survey of the user's needs must be fully documented and, in all ways, found thorough and consistent regardless of the individual's personal style and approach to a problem Third, the "requirements" or preliminary plan must be formulated as a result of the joint efforts of the user and the systems analyst to communicate with each other and, through this document with the rest of the organization

A data systems request form such as the sample in Figures 2 and 3, should be required so that user statements have a homogeneous format which assumes completeness in their thought processes prior to the investment of systems manpower to the request The systems department should respond to every user request with a standardized form similar to the one in Figure 4, which appraises the qualitative and quantitative elements of the request and recommends a course of action.



DATA SYSTEMS REQUEST (A)

FROM DEPT OR DIVISION LOCATION

DATE ____________ NEW SYSTEM □ CHANGE □ DSR NO ____________

SYSTEM NAME ____________

DESCRIBE REQUEST ____________

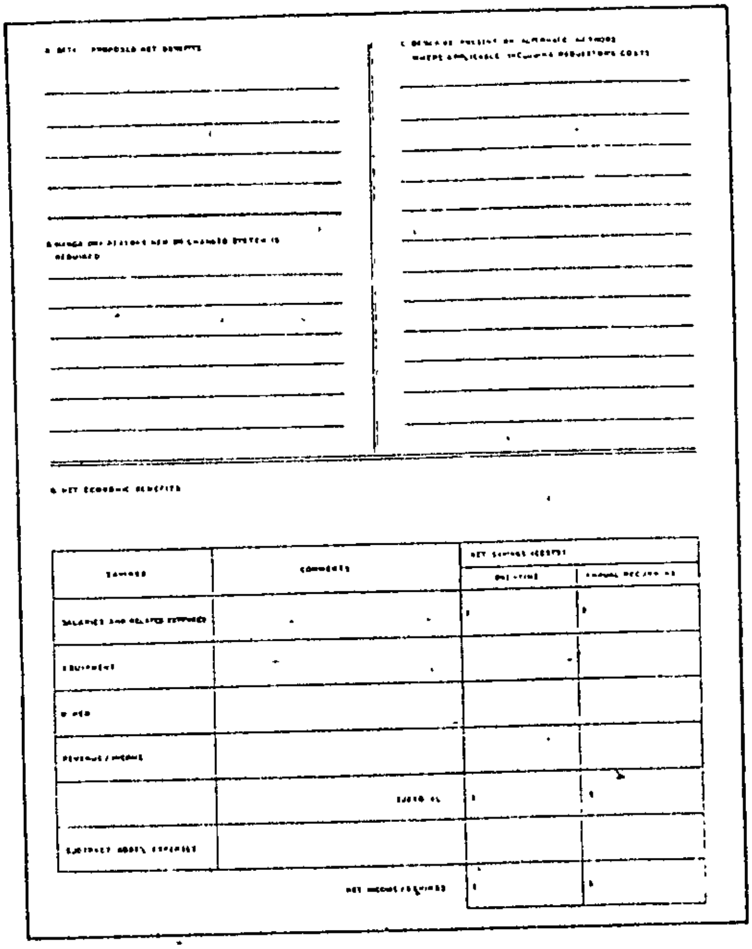
DESIRED IMPLEMENTATION DATE ____________ SIGNATURES

____
[illegible]

____
[illegible]

Figure 2. Data Systems Request Form (front)

If there is a recommendation to proceed, the survey of user needs must be carefully prescribed to include a detailed analysis of the present system all functions, objectives, inputs, outputs, files pertinent tasks, manual and machine operations, formulas, tables controls, and interfaces with other systems The proposed system must be defined in a similar manner but should also include such elements as detailed descriptions of the *cnanees* between the present and the proposed systems all alternatives which could satisfy the objectives of the proposed system constraints which are inherent to the proposed system (e g , specialized hardware availability of input etc), plus user criteria and justification for the new system



[illegible]

[illegible]

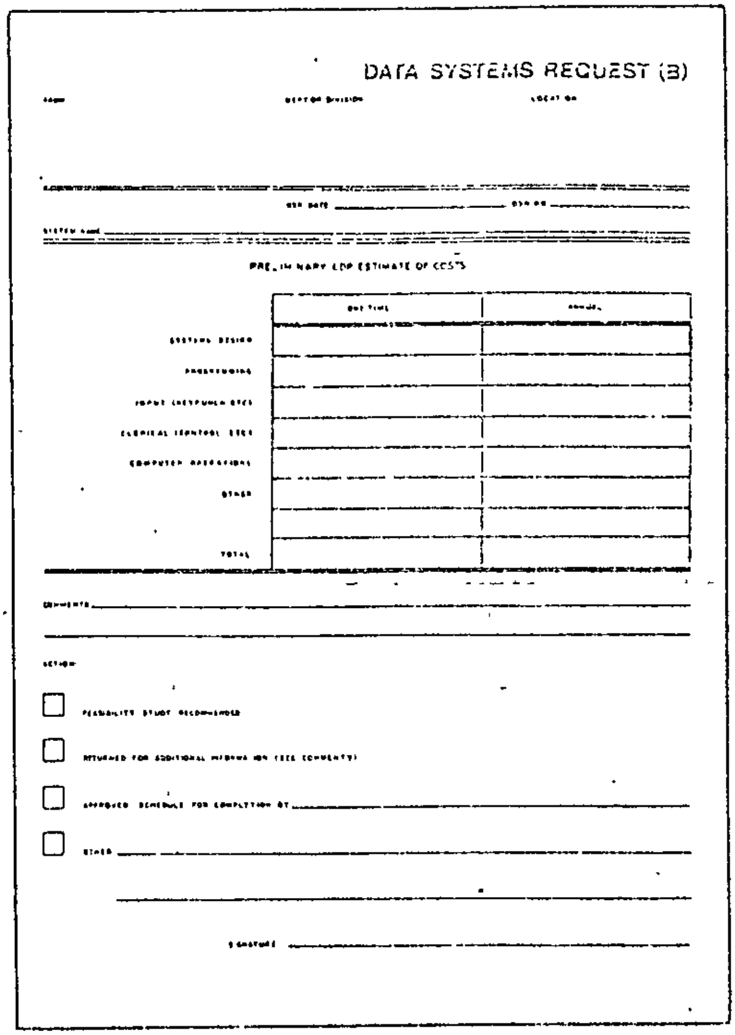
[illegible]

[illegible]	COMMENTS	[illegible]	
		[illegible]	[illegible]
SALARIES AND RELATED EXPENSES		$	$
EQUIPMENT			
OTHER			
REVENUE/INCOME			
	[illegible]	$	$
[illegible]			
	[illegible]	$	$

Figure 3. Data Systems Request Form (back)

Functional Specifications

These specs must be developed to arrive at a time manpower, and cost estimate of the proposed system before there is a further commitment of resources to the project This activity must include very precise and exacting analyses of all the elements previously defined Such analyses could be based on systems flowcharts record layouts report layouts, file and table layouts, detailed definitions of all calculations, and system algorithms From these will come workload projections manpower estimates, machine cost estimates and other economic considerations which should be reported in a format similar to those shown in Figures 5 and 6



DATA SYSTEMS REQUEST (B)

FROM DEPT OR DIVISION LOCATION

REC DATE ____________ DSR NO ____________

SYSTEM NAME ____________

PRELIMINARY EDP ESTIMATE OF COSTS

	ONE TIME	ANNUAL
SYSTEMS DESIGN		
PROGRAMMING		
INPUT (KEYPUNCH ETC)		
CLERICAL (CONTROL ETC)		
COMPUTER OPERATIONS		
OTHER		
TOTAL		

COMMENTS ____________

ACTION

□ FEASIBILITY STUDY RECOMMENDED

□ RETURNED FOR ADDITIONAL INFORMATION (SEE COMMENTS)

□ APPROVED SCHEDULE FOR COMPLETION BY ____________

□ OTHER ____________

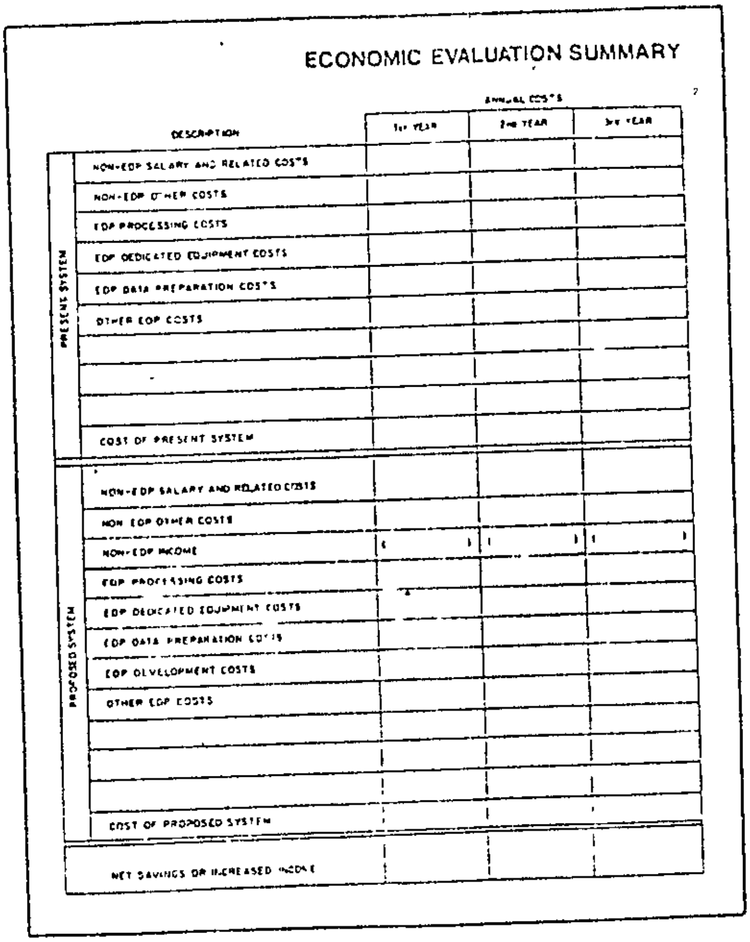
SIGNATURE ____________

Figure 4. Data Systems Response Form

These functional specifications will allow a final analysis of the "feasibility" of the proposed system its potential benefits, its inherent limitations its costs its time frame, and its total simplicity or complexity Most important they provide a final sign-off document which forces the user to verify that every conceivable base has been touched in an effort to understand his request and communicate the consequences of that request to him and the rest of the organization before work is begun

Systems Specifications

These specs must be fully defined, as described in the preceding paragraph However, there must be an eye toward the *efficiency* of the proposed system rather than its *economics* There must be standards for all flowcharting techniques and symbols, programming language selection, data file designs, manual procedures (in the systems development and system operation processes), system controls processing logic, as well as test files and procedures All these standards produce the final design of the system as a whole



ECONOMIC EVALUATION SUMMARY

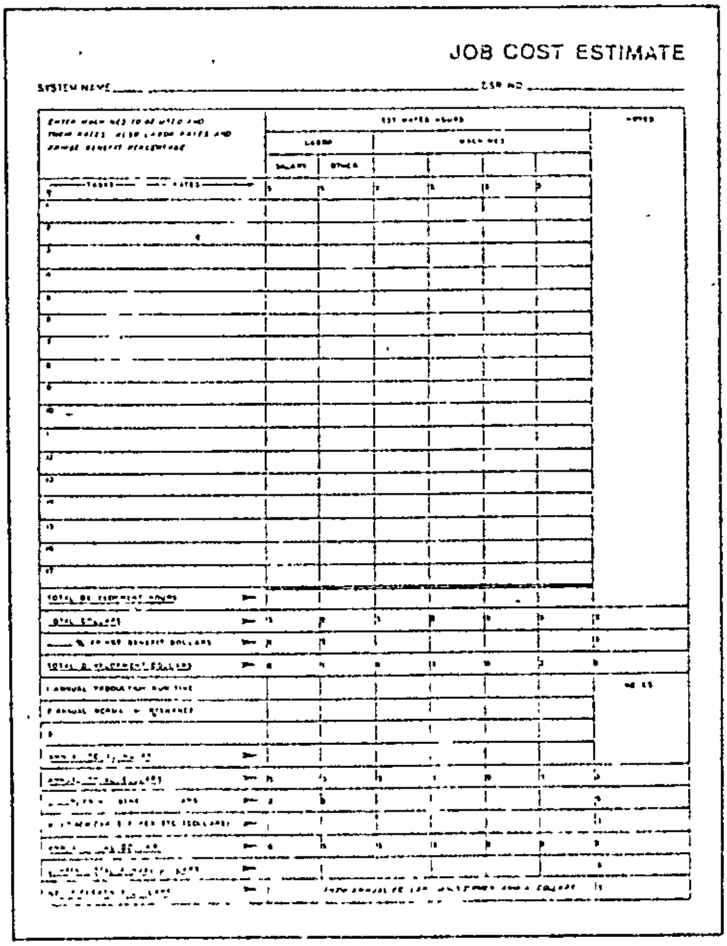
		ANNUAL COSTS		
	DESCRIPTION	1st YEAR	2nd YEAR	3rd YEAR
PRESENT SYSTEM	NON-EDP SALARY AND RELATED COSTS			
	NON-EDP OTHER COSTS			
	EDP PROCESSING COSTS			
	EDP DEDICATED EQUIPMENT COSTS			
	EDP DATA PREPARATION COSTS			
	OTHER EDP COSTS			
	COST OF PRESENT SYSTEM			
PROPOSED SYSTEM	NON-EDP SALARY AND RELATED COSTS			
	NON EDP OTHER COSTS			
	NON-EDP INCOME	( )	( )	( )
	EDP PROCESSING COSTS			
	EDP DEDICATED EQUIPMENT COSTS			
	EDP DATA PREPARATION COSTS			
	EDP DEVELOPMENT COSTS			
	OTHER EDP COSTS			
	COST OF PROPOSED SYSTEM			
	NET SAVINGS OR INCREASED INCOME			

Figure 5. Economic Evaluation Form

Program Specifications

Though all program specs must be standardized, this area offers the fewest effective tools Every experienced manager in data processing eventually acknowledges that the majority of his problems could be eliminated if he had adequate programming standards to impose on his organization

The purpose of programming standards is to apply detailed restrictions and guidelines to the process of programming Standards should include the languages which may or may not be used the allowable uses



JOB COST ESTIMATE

SYSTEM NAME ______________________ CSR NO ______________

[illegible]

Figure 6 Job Cost Estimate Form

of various facilities such as the console typewriter, required data validation steps, blocking factors, documentation factors (date in julian or other format required month, day, last business day, number of days in month, other fillers) guidelines for spooling checkpoint/restart, partition sizes, and use of return codes Additional standards must specify such topics as printed output formats for controls headings and special forms In the Job Control Language (JCL) there must be clearly defined naming standards for job, step module, data set, and the other breakdowns, and coding must be standard for volumes, input/output class assignments and priorities, job class assignments tape labels, and other areas

There must be standards for program documentation used subsequent to the programming activity (e g , for maintenance, modifications, and conversions) Such documentation should be standardized with regard to a narrative which includes program identification, purpose special features, special requirements, and controls Logic flowcharts should be complete, consistent and cross-referenced to the source language program All techniques and tables should be clearly_specified and illustrated, including randomizing formulas search techniques, complex table structures, as well as external and internal logic switches All input/output layouts must be specified, indexes and control fields described, sort sequences specified, and other items delineated Especially important is the need for clear descriptions of files which are used in multiple programs of applications These files must be clearly defined within the application as being input or output for each individual program Source and object listings must be complete with data division and procedure division maps, ,comments, cross-references and JCL cards (including / /LBLTYP VOL, DLAB, XTENT, and TPLAB cards) in 80-80 listing.

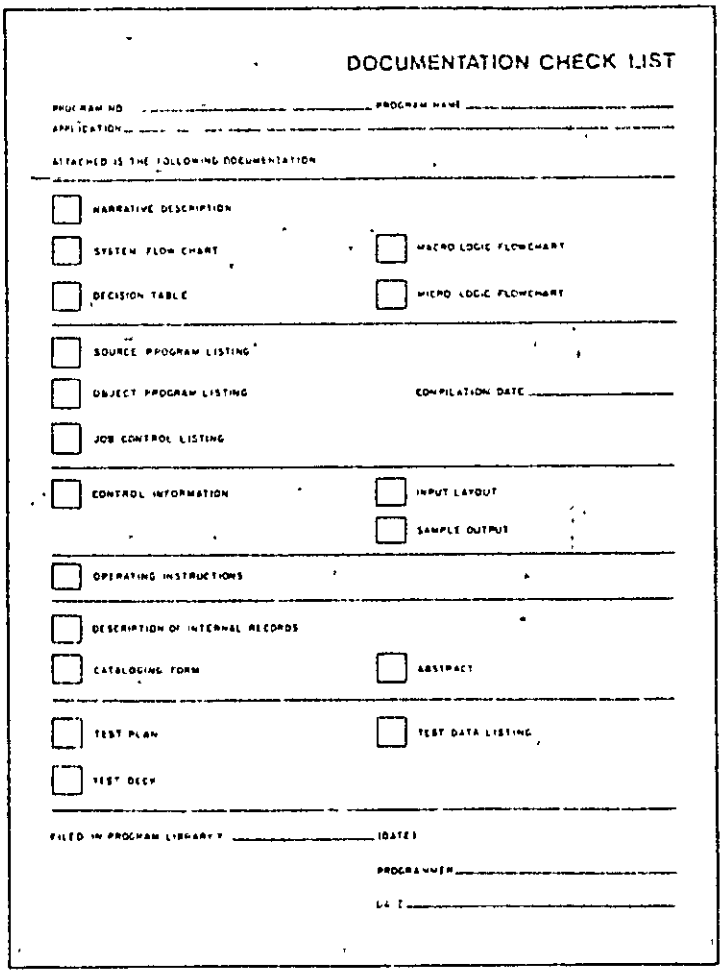
Last and most important, detailed information about controls is required This material must include information on data preparation, transmission from user to DP department, conversion of data to machine-readable form, creation and updating of master files, transmission of data from program to program, and handling of errors and exceptions Standards should require controls to be consistent, explicit, and easily auditable A run book should be defined in standardized format with operating procedures and guidelines, testing data and procedures, and all live and dummied data files which were used to test the program

Figures 7, 8, and 9 are good examples of standard documentation forms for the various steps mentioned

As an aid to creating systems and programming standards, Hierarchy plus Input, Process and Output (HIPO) (see portfolio 4-01-05) documentation should be developed to relate various programs to each other within each system Such standards are usually ignored because of time constraints and lack of knowledge regarding their value If properly done, HIPO standards and all the previously described system design and development standards become synonymous, and diminish the entire problem of developing standards that people will understand and follow

Briefly, HIPO involves a Visual Table of Contents (VTOC), which shows the functional structure of the system, the relationship of all diagrams, and the contents of each diagram There are overview diagrams to introduce various functions listed in the VTOC, and there are directions for detail diagrams to explain the process supporting each function, the inputs and outputs relating to that process, and cross-reference to routines, labels, and flowcharts within the system

HIPO takes what can be a confusing process of tables, charts, and layouts, and blends the parts into an interesting, logical story about the



DOCUMENTATION CHECK LIST

PROGRAM NO ______________ PROGRAM NAME ______________

APPLICATION ______________

ATTACHED IS THE FOLLOWING DOCUMENTATION

□ NARRATIVE DESCRIPTION

□ SYSTEM FLOW CHART □ MACRO LOGIC FLOWCHART

□ DECISION TABLE □ MICRO LOGIC FLOWCHART

□ SOURCE PROGRAM LISTING

□ OBJECT PROGRAM LISTING COMPILATION DATE ______________

□ JOB CONTROL LISTING

□ CONTROL INFORMATION □ INPUT LAYOUT

□ SAMPLE OUTPUT

□ OPERATING INSTRUCTIONS

□ DESCRIPTION OF INTERNAL RECORDS

□ CATALOGING FORM □ ABSTRACT

□ TEST PLAN □ TEST DATA LISTING

□ TEST DECK

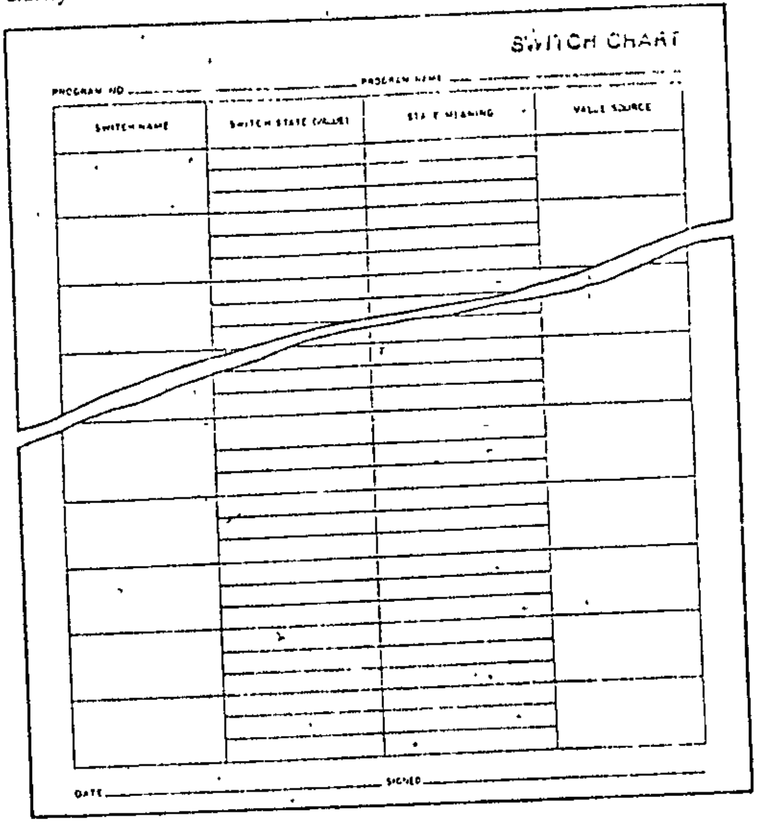
FILED IN PROGRAM LIBRARY? ______________ (DATE)

PROGRAMMER ______________

DATE ______________

Figure 7 Documentation Check List Form

system The HIPO technique can be valuable in teaching the DP staff about an existing system, in helping them design and develop new systems, and helping management monitor systems design and programming activities, estimating workloads more accurately, simplify testing, clarify interfaces, and aid in program and systems maintenance



SWITCH CHART

PROGRAM NO ______ PROGRAM NAME ______

SWITCH NAME	SWITCH STATE (VALUE)	STATE MEANING	VALUE SOURCE

DATE ______ SIGNED ______

Figure 8. Switch Chart Form

OPERATIONS STANDARDS

Operations has the most difficult job in the data processing organization because it must show the results of all the activity which preceded "live processing" When things go wrong operations is the visible tip of the iceberg, so the user and upper management first attack there If enough pressure is applied operations may end up solving systems and programming errors (as in the cartoon illustration of the swing) Standards for computer operations is, therefore, the last line of defense before a system is "completed" and it is especially important that there be clear standards for a variety of activities in this department

Operational Features Checklist

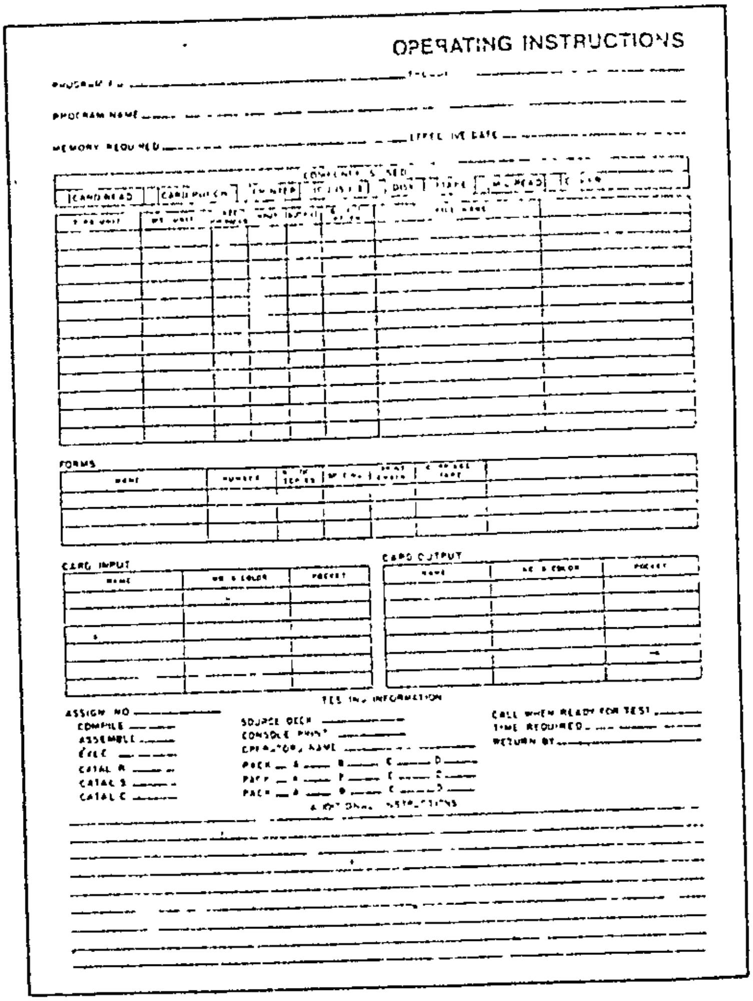
This requirement, which should be fulfilled by the programming department, answers the following questions

(1) Is all variable information passed to the program through the JCL so that operator decisions are not required?
(2) Do programs that process sorted data check for sequence?
(3) Are control totals provided for each step?
(4) Is control information passed from program to program?
(5) Do the programs maintain and verify batch totals and record counts?
(6) Are all programs 80K bytes or less?
(7) Are overlay structures used to reduce core requirements?
(8) Do the programs accommodate changes or expansions without requiring recompilation?
(9) If more than one on-line date or control card is necessary, is entry restricted to one given step?
(10) Do all master file records have expansion bytes?
(11) Will all tapes be limited to one data set per reel?
(12) Are printed reports designed for standard form paper and standard carriage tape?
(13) Do all report headings have a page number, meaningful title, and DSNAME?
(14) Is a print data set that is destined for special-form paper preceded by several "dummy" pages for ease in form alignment?
(15) Is invalid data rejected or flagged and an error report produced?
(16) Is input data validated for numerics if arithmetic operations are to be performed on them?
(17) Are any output data sets intended as input for another system?
(18) Are there any private disc packs maintained by another system?
(19) Is the estimated execution time per job stream less than 60 minutes?
(20) Is the checkpoint facility used in programs than run ≥ 15 minutes and/or process ≥ 1 reel of tape?
(21) Is the data retention specification adequate to allow the rerunning of the last two cycles of any procedure at any time?
(22) Are restart procedures clearly documented for each job step?
(23) Does each ABEND operate upon detection of an error, and is it fully documented in the run book?
(24) Are return codes used?
(25) Are cataloged procedures used wherever possible?
(26) Are standard labels used to ensure data set integrity and date protection?

Standard Computer Operator Documentation

Documentation guidelines should include

- Run books with proper standards
- Systems flowcharts
- Lists of source programs object programs and load modules
- Lists of any private disc contents



OPERATING INSTRUCTIONS

PROGRAM I.D. ____ [illegible] ____

PROGRAM NAME ____

MEMORY REQUIRED ____ EFFECTIVE DATE ____

COMPONENTS USED

□ CARD READ □ CARD PUNCH □ PRINTER □ [illegible] □ DISK □ TAPE □ MICR READ □ [illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	FILE NAME	

FORMS

NAME	NUMBER	[illegible]	[illegible]	[illegible]	[illegible]	

CARD INPUT

NAME	NO. & COLOR	POCKET

CARD OUTPUT

NAME	NO. & COLOR	POCKET

TESTING INFORMATION

ASSIGN NO ____
COMPILE ____
ASSEMBLE ____
EXEC ____
CATAL A ____
CATAL B ____
CATAL C ____

SOURCE DECK ____
CONSOLE PRINT ____
OPERATOR'S NAME ____
PACK ____ A ____ B ____ C ____ D ____
PACK ____ A ____ B ____ C ____ D ____
PACK ____ A ____ B ____ C ____ D ____

CALL WHEN READY FOR TEST ____
TIME REQUIRED ____
RETURN BY ____

ADDITIONAL INSTRUCTIONS

Figure 9 Operating Instructions Form

- Space requirements or non-private disc packs
- Lists of all tapes and their retention periods
- Lists of all programs with special core requirements
- Summary of distribution of all reports
- Back-up and trouble-shooting procedures
- Explanations of the use of temporary and passed data sets and utility programs used (for restart)
- File organization description for each data set
- Samples of diagnostic messages with explanations
- Full explanation of all special date and control cards

Acceptance Criteria

Acceptance of the run book, i e , documenting a job stream of more than one program, must be contingent upon a well-defined set of criteria This book provides instructions for operations personnel in scheduling, setting up, and running a given job while the programmer has responsibility for setting it up and maintaining it The programmer's guidelines for documentation standards were covered previously, however, the operations manager has final responsibility for accepting or rejecting the run book produced, and it is vital that he have high standards for this important document

Briefly, the run book should have the following major sections

- Job Description -- includes all functions, machine configurations and flowcharts
- Operation Sheet — for use under the operating system, to track changes from run to run
- Job Stream Management — explains JCL cards and JCL logic
- Card Layouts — for all JCL cards
- Checkpoint/Restart — for use in ABEND at each step in the system
- Unit Record Data Sets — for control use if unit record steps are involved
- Input/Out Samples — detailed explanation of all input and output

Card input must be limited to 3 000 cards before a card-to-tape run should be required

There should be an elaborate checklist which forces the operations manager or job acceptance specialist to critically scrutinize each run book for completeness, coherence, usability under duress, and clarity to posterity The run book is the ultimate documentation for every system, thus, only to the extent that it is thorough is system documentation complete

Job Acceptance and Turnover

It should go without saying that every facet of the run book should be systematically tested by operations before being accepted as valid During the testing phase of systems development, programmers are looking for bugs in each program and the overall system At this time, operations should be validating all the documentation and determining whether the system is acceptable.

When the Job Acceptance run is scheduled, the job is ready for final review by operations personnel, and there should be carefully defined procedures for the criteria it must meet For example, a job acceptance specialist should be present together with an I/O controller and representatives from operations systems, and programming The run book must be employed, and all "redo" and "restart" procedures must be tested *by operations personnel* not programmers

Job Turnover should be defined as a formal procedure after successful completion of a Job Acceptance run.

USER STANDARDS

Besides having documentation for use by programmers and operations personnel, the system must also include documentation and support materials to assist the user The Standards Manual must therefore, include tutorial, instructional, or reference materials specifically designed for the non-data processing people who will interface with the system Without standards for these documents, an otherwise well-designed system may fail because the user cannot provide clean data or effectively interact in some other way with the data processing department The documents required by users can be similar to those already mentioned (e g , card and file layouts, error conditions, flowcharts, or action sequences, explanations of forms, and proofs aud controls)

Some documents must include instructions on how to fill out forms or how to complete a transaction within the system. Users also need task descriptions, in a step-by-step format, complete with illustrations In addition, there must be managers' manuals which specify all functions, relationships with other groups, emergency procedures, and audit trails All these elements must be specified by definite standards Included may be job descriptions of various functions within data processing so that users will better understand the responsibilities of those with whom they must *communicate* effectively

STANDARDS MANUAL ORGANIZATION AND IMPLEMENTATION

There are probably an infinite number of ways to organize a good standards manual Every reader of this portfolio should plan a format that is most suitable to his organization In essence, pick a plan that is most likely to harmonize with the personalities who are expected to use the manual because *effectiveness* is more important than elegance

Every manual needs a format Manuals which are likely to be frequently updated should be looseleaf, and have page and section numbering schemes which enable easy additions and deletions It is also a good idea for each page to have the organization name, issue date, revision date, chapter number, section number, and page number

The organization of the manual itself should be simple since, in many ways, its contents will contain many complexities At the least, it should have a table of contents, an introductory section (on purpose, method of usage, span of coverage, philosophical orientation, and other such topics), a section for systems and programming, a section for computer operations, and a section for non-data processing users and managers It is also helpful to elaborate with other sections on controls and input preparation; a glossary of terms will increase effectiveness and usability

Most organizations have *some* existing standards Therefore, when planning a standards manual build it around whatever exists so that it is not totally new and different Too great an upheaval will neutralize any contemplated benefits from a standards manual because there may be hostility and a long learning process for everyone in the organization

RECOMMENDED COURSE OF ACTION

At the outset, analyze the problems in your organization and try to determine, in outline form the specific variances between your environment as it is and the ideal situation you envision What would it take to give your organization all of the characteristics of a well-run data processing shop? Next, compile all of the standards which already exist in any form, (good and bad) Third, design a general outline of the standards manual you think should be used (i e , write the table of contents) and then group your existing standards according to that outline This will give you a beginning framework for each section

Finally, treat each section (or chapter) as an individual manual unto itself, and proceed to develop comprehensive standards in coordination with experts from each department or functional area Build on what is recognized to be good and discard what is bad Try to keep the formats among sections consistent even though many different people will be contributing

Once complete, the manual must be accepted by managers in each of the functional areas it purports to standardize, and the new rules or format protocols must be imposed on all new work done as of a specific date The standards should be sufficiently flexible so that *no* variance will be tolerated by senior DP management

Provision for updating (where consistent pressure for variance indicates the need for modification) must be defined, and each functional area must be required to review its portion prior to a periodic revision process In this way, the standards manual will always be practical, it will also be in close touch with reality and the immediate needs and abilities of the organization

This portfolio was written by Lawrence Richman

STANDARES

AUTOR:	L. WILSON	CLAVE:	RF-2
TITULO:	THE DOS AND DON'TS OF DOCUMENTATION		
IDIOMA:	INGLES	TIPO DOC.:	ARTICULO
REF.:	DATAMATION SEPT. 81		

by Lindsay Wilson

The meeting had just started and already the man was the picture of despair

"That's it!" he said, pointing accusingly at a four-foot high stack of printout "That's the documentation the software company sent when we bought its package four years ago You think anybody's ever read it? You think anybody's ever tried?"

The man was an executive for a major film corporation and was responsible for keeping tabs on worldwide film distribution He had no quarrel with the software package itself. The senior programmers had been making quick, successful modifications to it ever since it had been integrated into the system The package had delivered what had been promised, and the programmers had been able to expand and manipulate it without stumbling over the software.

But the corporation's department lines had been redrawn Now that each European branch was to be responsible for its own distribution, the executive had to ship large sections of the home based dp system overseas, with explanations of course

"Send the Europeans that stuff," the senior programmers said, referring to the man-sized pile of paper The programmers could not be coerced into further documenting the system The 175 pounds of explanations had been on shelves in the data center, had been moved to a side room, had migrated unused through several floors and had finally come to rest in the conference room

' The new software marketing firm had lots of office space and four pieces of furniture But its contracts were in order because it had customers who had already paid for the firm's acounting package The authors of the system presented a thoroughly documented flow of programs that marched through general ledger accounts receivable, accounts payable, and cost accounting and produced summaries and reports along the way

"If the documentation doesn't cover it," said one author of the package "my partner or I can explain any routine in any module you pick out " They knew their material cold

"The time period accounting people have been screaming for years," the systems designer at the television news corporation said "Their data processing was set up in—get this—1966, and the system always worked It requires almost no maintenance But the way things were done in '66 has gotten so time and core consuming that redesigning that system is now my top priority "

The COBOL system in question included such quaint touches as having daily edit and audit reports sent halfway across the city to be hand corrected by the accounting department The hand corrected copies were sent back to the data center, cards with corrections were punched and the edit/audit was run again and again until the run was error-free

The classic definition of "documentation"—1) supplying of documents or supporting references use of documentary evidence, 2) the documents or references thus supplied 3) the collecting, abstracting, and coding of printed or written information for future reference—would cover computer language reference books, most written explanations accompanying hardware, almost any book that IBM puts out, green and yellow cards dictionaries of dp terminology, and in short, everything dp people use for reference that they do not generate themselves

What about the documentation companies produce themselves? Just a mention of the word brings groans of boredom

The existence and organization of documentation varies enormously from corporation to corporation and often from department to department But the larger and more complex a system is the bigger the job I consider any documentation produced in-house to be "live," that is subject to the same maintenance revisions and even redesigning that the system itself goes through Documentation coming in from the outside, such as PL/1, language books, etc , is "reference "

If "reference" documentation is long overly detailed and somewhat heavy, that is not unexpected It is somewhat in the nature of PL/1 language books to be boring and for IBM software manuals to be excruciatingly detailed "Live" documentation, however, is meant to be used, not simply referenced If it's cumbersome or difficult, it's no good Some types of "live" documentation will get used far more often than others, but the principle remains the same dog-eared is beautiful

HOW TO DOCUMENT IN-HOUSE

Documentation produced in-house may include variations on the following

System documentation specifications, modular flowcharts, file layouts, report log and report samples system narratives, JCL logs, program [illegible] logs

Program documentation source documents, source listings program narratives, summaries of program logic, listings of program subroutines, user guides

Operations documentation run books, run logs, file libraries instructions on report balancing

Software subroutines processing summaries catalogs

It would be ideal if all categories were covered in a corporation's dp division and if a permanent staff were assigned to keep up with the maintenance of the documentation No more and no less is required, but in this still unenlightened decade gigantic corporations try to make do with one documentation consultant or a small staff of technical writers

Going back to the film executive with the unwieldy software documentation whoever bought the original package was doubtless impressed by its sheer poundage It was probably assumed that there would never be any problems

After having established that he had been sending out portions of the film distribution system in modules of five to 10 programs for an on-line system, we told him to forget his existing documentation and to stick with user guides Like many dp managers, he did not realize that there are many ways to document and that a standard user guide minimizes length and detail

For his purposes users by definition did not have to know anything about software or programming logic Since systems and programming authority remained in New York the European accountants auditing the film distribution would be performing easily controlled updating and balancing

What are the points illustrated by this scenario?

1. ''Documentation'' is extremely varied

2 Size means nothing Examine closely any documentation for hardware or software If you can t follow it, be ar in mind that you ll have to have it rewritten if you ever need to explain the system in detail to anyone

3 Keep track of modifications and add them to the existing documentation If you ever have to explain the system to anyone and the maintenance programmers are no longer there, you re in trouble

4 Stop everything and shop around if confronted with 175 pounds of paper

SAMPLE ACID TEST FOR DP MANAGERS

If given a week s time could you answer these questions on systems and programming?

1 How many systems do you have? How many programs in each system?

2 What languages and hardware are involved?

3 How many off-line jobs/job streams/ production runs are executed per day, per week for each system?

4 How many reports ar produced? How many of these can be e ated?

5 Without notice, can you demonstrate (with charts and reports, or on line) any and all of your systems to a top executive from an outside division?

6 If the auditors show up today, are you ready?

7 Is your system affected by obsolete methods or material which can be eliminated from processing?

8 Do you know when all the files in your system are updated?

9 Do you know, at least in paragraph form what each program in your system does?

10 Do you know what each of your programmers did today or this week? Is anyone keeping track?

If you can t answer at least half the questions positively, you are headed for problems Up-to-date documentation is every system's safety net Don't wait until you're buried in paper, relying heavily on maintenance programmers at $300 a day or you're given a deadline by the auditors

USER GUIDES AND MAPS

The budding software company knew exactly what it needed for its system a user guide The company described its typical user very precisely and demonstrated that it didn t need program documentation comments, file layouts, cross-referencing of routines, report layouts, and narratives were all in place for 50 interlocking modules The company wanted a writer to extract everything that would directly concern a user at a terminal, and to put it all into manual format for marketing

Companies in this situation should be aware of five rules when soliciting a writer

1 Ask to see samples of his work

2 Define the user's needs exactly and avoid wasting time by going off on a tangent (Is the user guide also a training manual? If so, more detail is called for, and this must be decided on at the beginning)

3 Establish the level of detail you want For example will all possible error messages be listed or only those to which the user can respond?

4 Establish the tone or style of writing

5 Don't count on the writer for details such as formatting and paragraphing these are usually done by a typesetter

In the third scenario, that of the systems analyst required to redesign the ancient time period accounting system, the need was for ''temporary'' documentation, compromise in favor of time and effort Normally, we would go through each listing, producing summaries of program logic and listing the subroutines and the use of any in house software routines We would have source doc ments and report samples for each progra and we would provide file layouts, narrative and, finally, a flowchart showing the wh system

But wanting to avoid unnecessary d tail of any kind, we started with a map of t system that underlined every repetitive aspe of processing every laborious card-to ta procedure, every old sort-and-merge pr gram We concentrated on revealing eve thing that appeared redundant or overly lo and involved Most of the tape files had virt ally identical formats, and extracted five layouts instead of 18 Samples of all 42 ports, but not the program logic that produ them, were provided (The system had und gone minimal maintenance and we assum that endless move to'' statements were of interest) The systems analyst was th able to pinpoint any questions he had, a further documentation consisted of listing t calculations made in the reports and noti when and where updating occurred

A note on redesigning a system wi out documentation many corporations ha sleeping systems—no maintenance, no tr ble, no documentation just old The disc ery that nothing is written need not be reas for panic, but a technical writer with so systems background is probably requir The priorities are to keep a broad view of system and to avoid most detail

In deciding whether or not your s tems need documentation, there is more volved then trying to export, market or re sign segments of processing

Lindsay Wilson is the *nom do plume* of the manager of a team of docume tation specialists and systems analys who work under contract for several major corporations in the northeast U S

20

Some Practical Experience with a Software Quality Assurance Program

G. G. Gustafson and Roberta J. Kerr
Computer Sciences Corporation

SUMMARY: Within a production programming environment, a software quality assurance program (QAP) was instituted to produce standards, conventions, and methodologies for all phases of the software development process. Software language standards and several support processors, in turn, developed. The authors offer a plan which may help others avoid some of the pitfalls they experienced while attempting to construct a meaningful software QAP.

1. Introduction

The typical production[1] environment has certain characteristics which may negatively effect software quality:

—Personnel of widely varying levels of skill.

—Small (usually one-man) project staffs that consequently increase the span of the project manager's control

—Software-naive customers who are usually interested only in software output.

—Poorly defined but often highly complex customer objectives.

—A high turnover rate for personnel, resulting in the continual review of existing software and high training costs.

—Externally or internally generated constraints (e.g., cost, time, and manpower limitations).

—Hardware complexities which occasionally force the applications programmer to operate as a systems programmer, not directly working toward the actual goals outlined by the customer.

—Poor quality of existing programs that were produced without the benefit of coding standards, configuration management techniques, or proper documentation. Many such programs require frequent modification, are costly to exercise, and result in serious difficulties in maintainability, reliability, and internal consistency.

A software quality assurance program (QAP) cannot attempt to resolve all of these problems at once; rather, resolution of the problems with the greatest impact should be its immediate objective. Regardless of its long-term goals, a successful QAP must at least have the following characteristics:

Feasibility: The program must be both technically and managerially feasible. It must appeal to the instincts of the technical staff and address management's immediate time- and cost-related concerns.

Currency: The program must be based upon current methodologies. It must not be allowed to become outdated, no matter how good the staff perceives the original version to be.

Extensibility: The program must be easily extended to include new software methodologies.

In a production programming environment (see Figure 1 for an idea

CR Categories and Subject Descriptors D 2 9 [**Software Engineering**]: Management-*software quality assurance*, J. 1. [**Computer Applications**]. Administrative Data Processing-*military*
General Terms . Standardization, Management
Additional Key Words and Phrases software development, production programming, military standards

Author's present addresses G G. Gustafson, Computer Sciences Corp, 443 Inyokern Road, Ridgecrest, CA 93555, R J. Kerr, Computer Sciences Corp, 4045 Hancock Street, San Diego, CA 92110.

[1] We use the term "production" as an adjective, as in "production environment" or "production programming" With it, we are distinguishing between programming performed under cost and time constraints for a customer who is usually a nonprogrammer and all other forms of programming (e g, hobbyists, research, etc).

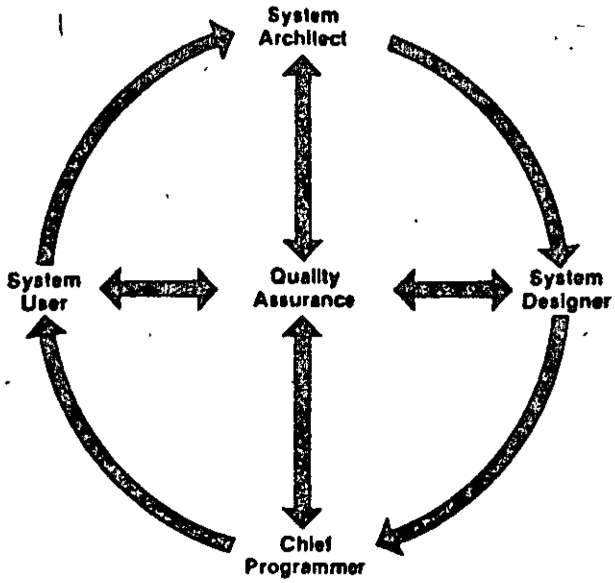


Fig. 1. Interrelationships Within an Ideal Production Environment.

of the interrelationships involved), a QAP must support the development of software that meets, or exceeds, the customer's expectations; is easily maintained; and is delivered on time and within cost. Furthermore, even a well-intentioned software QAP will fail unless pursued tenaciously. It is imperative that a software QAP be sufficiently staffed with the most highly qualified technical personnel available. Because of the negative ramifications of failure, it is better to rely upon existing programming methodologies than to initiate an inadequately staffed and/or funded QAP.

2. Establishment of a Realistic Quality Assurance Program

At the very first, the QAP must be documented. This document should focus on the intended scope of the program. Quality assurance implies many things to many people. A program plan should be bounded by the areas of concern and defined goals. Our program plan addressed many issues, including organization, procedures of operation, and a very ambitious list of planned QA activities.

Managers contemplating the establishment of a QAP must be aware of one potentially significant interference. If the recommended staffing is followed, the QAP will draw the very best personnel into the program itself. Obviously, these senior technicians will be in high demand for "firefighting"; however, as members of the QAP, they should not be reassigned, even for a relatively short duration. QAP members should only be considered an available consulting staff. Although a portion of their time will be spent in consultation not directly supporting the QAP, staff members will become familiar with a wide range of software development problems (some of which may be correctable through their efforts in quality assurance).

With these highly qualified individuals playing the dual role of quality assurance technicians and general consultants, the organization gains two benefits. First, a technically sound QAP exists, supported by personnel who are not totally removed from the production environment. Second, consultation activities provide QAP staff members with a certain satisfaction otherwise missing in their line of work.

The Chief of Software Quality Assurance

Considerable thought should be given to the choice of a chief of software quality assurance. ...
gest that the candidate posse
following qualifications and ch
teristics:

Technical experience: Sinc
QAP chief will interface with a
variety of project types withi
production programming env
ment, he should be a senior
member who has had technic
opposed to administrative) p
management experience.

Technical currency: He mus
successfully used current pro
ming methodologies (e.g., struc
programming, top-down desig
implementation, egoless pro
ming, etc.).

Willingness: Because a
large amount of time will be
managing a team of other
technicians, the chief must be a
ing leader (i.e., the favored can
should first be invited to consid
job before he is actually chos
the position).

Communications skills: Sin
QAP chief must deal with ma
ment, sometimes forcefully, he
have verbal prowess; becaus
quality assurance staff will pr
written standards, policies and
cedures, the chief must have
writing skills.

Once a chief of software q
assurance has been selected, th
of program development and
agement should become his r
sibilities.

Initial Tasks Within the Pr

There are five preparatory
which should be performed b
chief of software quality assu
and can be accomplished witho
ditional staffing. The results of
initial tasks must be approve
wholly supported by manag
for they form the basis of the
quent QAP. Failure to elici
management support will un
edly result in the failure of the
itself.

The first task to be unde
by the QAP chief is the rev
existing software quality ass
programs within his own org
tion and others. Unlike many
areas, the software quality ass

programs of most organizations are open to scrutiny by competitors. This review will provide a measure against which the chief of software quality assurance can assess his own projected organization. Two such programs include those conducted at the Jet Propulsion Laboratory [4] and the Naval Ocean Systems Center [3]. Although the literature on QAPs is still far from extensive, [2] may be of particular interest.

Second, it is essential to define the development life cycle used (or preferred) within the organization sponsoring a new quality assurance program, based upon the type of software services performed. A software development organization differs from a software maintenance organization not only in the end product, but also in the development cycles required to complete a task. It may be necessary for an organization to adopt two or more "standard" development life cycles if it contains both development and maintenance within its contract base.

Third, once a software development life cycle has been chosen, the deliverables for each phase must be identified. In a multi-life-cycle environment, each life cycle has its own deliverables and should be considered as if the other cycles did not exist. The deliverables for a five-phase software development life cycle might be those displayed in Figure 2. Although additional deliverables may be required, these are generally sufficient in projects which strictly adhere to top-down design and implementation.

Job titles within a given organization generally reflect levels of competence. As his fourth task, the chief of software quality assurance must determine which titles are associated with which life-cycle phases (again see Figure 2). New titles rather than traditional ones may be needed to more fully describe an individual's role—"systems architect" or "chief programmer" as opposed to "senior computer scientist" or "senior member of the technical staff." As a rule of thumb, we have found that the earlier phases of software development require far more expertise than the later ones. That is, greater competence is required as one moves from implementation (coding) to design, and likewise from design to definition.

Finally, before proceeding further, the chief of software quality assurance must receive management approval of the suggested development cycles, their associated deliverables and proposed staffing.

Setting up a Quality Assurance Plan

Once the QAP chief has completed those five initial tasks listed above, two new members should be added to the staff: a senior technician who can assume the responsibilities of the program chief if necessary, and a clerk who is experienced in technical writing, word processing, or computer terminal operations. Next, internal procedures for the review and development of standards must be established. This problem may be resolved by forming two adjunct organizations: a technical review group (TRG) and a management review group (MRG).

The TRG whose members are drawn from the technical staff advises the chief of software quality assurance on technical matters, reviews proposed standards, and represents the technical staff itself. We have found that technical staff mem-

	Software Development Phase				
Personnel	Conceptual	Definition	Design	Implementation	Maintenance
Architect	——	——	- - - - -		
Designer		- - - - -	——	- - - - -	
Chief Programmer			- - - - -	- - - - -	- - - - -
Quality Assurance	- - - - -	- - - - -	- - - - -	——	——
Products	Functional Description ▲	Users Manual ▲ Definition Document ▲	Database Design ▲ Interface Specifications ▲ Software Design Document ▲	Computer Software ▲ Maintenance Manual ▲	Revised Documentation ▲

Fig. 2. Personnel and Products in a Production Environment (primary responsibility ——; secondary -----).

23

ters who are interested in joining a TRG usually contact the quality assurance staff on their own. A TRG should be composed of at least four, but no more than ten members. The number of TRGs established depends on the long-range goals of the QAP. The language TRGs (i.e., one TRG for each programming language used within an organization) are the most easily discernible. The number of documentation TRGs depends on the life cycle(s) and corresponding deliverables. The chief of software quality assurance should be the chairman of each TRG.

The MRG, whose members are selected from management personnel, acts as an internal reviewer for all TRG work products; it ensures that proposed standards are acceptable to both the company and the customer. The manager of the organization sponsoring the QAP should be the MRG's chairman, and he should select all members. The chief of software quality assurance and his assistant should automatically serve on the MRG.

The members of both the MRG and TRG benefit through a feeling of personal involvement in the standards-making process. If technicians feel that they have had some say in what standards are developed, they are more likely to support those standards in discussions with their peers. Likewise, if managers actively participate in standards review, they can better represent those standards to their customers who must eventually pay for their establishment.

After internal procedures have been agreed upon, the quality assurance staff should prepare a charter. It should describe the organization of the QAP and the responsibilities of each of its components; the internal procedures to be followed (including auditing and waiver procedures), planned areas of QAP involvement along with a tentative schedule, and the QAP's implementation notice. The senior manager of the sponsoring organization (i.e., chairman of the MRG) should sign such a notice, the date on which the QAP organization will come into existence must also be specified. The plan should be reviewed at least quarterly by the TRG and MRG to ensure that organizational misalignments and errors do not occur.

The QAP Effort and Resulting Products

The required skills of team members decrease as a software project passes from its first to last phase. It is our recommendation, therefore, that a QAP's initial efforts be directed toward the last software development life-cycle phase (usually implementation), while striving to increase the staff's basic technical competence. Once the coding phase shows marked improvement, the next preceding phase can be considered. In this way, a QAP's real impact is more readily ascertained. As maintainability increases, so too should productivity—although there may be times when it seems productivity is slipping.

A QAP's activities differ from organization to organization. They usually involve the review, rewriting, and enforcement of existing standards or conventions which affect deliverables. Sometimes, these standards must be compared on a line-by-line basis; any differences in content should be carefully noted because the quality assurance staff must produce revised standards when inconsistencies exist. Current standards are generally static and represent the state of the art at some past time. Perhaps the most disturbing result of reviewing existing standards is the observation that often standards are not only outdated, but also in direct conflict with one another. We have observed a growing realization within government agencies that myriad standards do not achieve their avowed purpose, and may in fact be the cause of a large number of unacceptable software systems. However, rivalries hamper the consolidation of standards. A recent government conference on quality assurance (no proceedings available) recognized that acceptable software standards often evolve from

those organizations which p
good, maintainable software.
have found that organizations w
issue standards are easily
proached and are usually quite
ing to modify their standards to
more rigorous requirements.

Although standards and con
tions development is custom
looked upon with some disple
by a quality assurance staff, it i
single most important and visib
sult of a QAP. It is therefore in
ative that the standards and con
tions be acceptable to the tech
staff, management, and the
tomer. The acceptability of prop
standards is easily measured b
ease with which they pass thr
review by both the TRG and M
It may also be wise during the
dards development process to
in mind a principle formulated
Ambrose of CSC: If a distinct a
tage over current practice c
shown, then the standard shou
considered for adoption; if no
advantage exists or if a suffic
large enough return is not rea
then the standard should not b
ther considered. Since the writ
a standard is insufficient as a
sure of its value, the ability (c
bility) to enforce the standar
automation can be used as a
mate of its worth. Standards
are unenforceable by auto
means are usually not follow
practice; this generally elim
a large number of otherwise
intended, but unessential
dards. Only in very few cases s
a standard, unenforceable by
mation, be allowed to remain
this test.

Finally, the tools whic
needed to make existing stand
adaptable to automation as p
must be developed. The fail
develop such tools is the major
of most current QAPs, and m
ten results not from a lack of i
but from a dearth of adequat
gram staffing. Therefore, add
technical personnel (whose s
sponsibility is the developm
software tools) must be added
quality assurance staff.

24

Costs of a Software QAP

The costs of a software QAP well exceed the direct labor costs of the quality assurance staff and the usual ODC, G&A, and overhead expenses. We estimate that once quality assurance requirements are imposed, the cost of an ongoing project can double. This increase stems primarily from the expense of project documentation: the direct labor costs for personnel to write the documentation and the direct labor and ODC costs for final delivery. Many customers are unwilling to pay the price of quality assurance regardless of the benefits. However, once a QAP is established, projects operating under quality assurance requirements should show a marked decrease in maintenance costs, now estimated to often be as much as 70% of the total project effort [5].

There are also other costs to consider. One such expense is training. No matter how obvious standards may seem, a certain amount of training will be required to familiarize personnel with them. During the time between their initial promulgation and eventual acceptance (or enforcement, if required), language standards are usually not followed. Some code must therefore be rewritten to meet the language standard. This problem may be alleviated by publishing the standard well in advance of its date of effectiveness.

The customer must not only be convinced that the benefits associated with a QAP are worth the cost, he must also feel that the QAP will provide him with real long-term savings. Accordingly, management must be willing to spend time with a customer providing convincing arguments and be prepared for the consequent loss of time and corresponding expense.

3. Reusable Software

Quality software, once written, is a valuable and reusable resource. Too often much time is spent redeveloping existing software because its availability is not common knowledge If quality software does exist and, in many cases, is reusable, then such software ought to be collected in libraries. The promulgation of reusable software policies and procedures is a means of achieving the goals of a QAP with the least cost and time expenditure. The burden this places on the QAP organization should not be understated. There is a great amount of work involved in both the establishment and maintenance of reusable software libraries. Furthermore, tools are required to aid programmers who are not members of the quality assurance staff in determining if software that meets their needs already exists. If a truly workable reusable software policy is to be added to a QAP, it is mandatory that the quality assurance staff be increased by at least one technical member.

There appear to be two inexpensive ways of identifying sources of reusable software within an organization: reviewing existent libraries which are used by many programmers and formulating and processing a questionnaire that asks programmers to submit useful routines to the QAP. The review of existent libraries is best conducted in relatively large, multiprogrammer environments. Due to the discipline required in larger projects, there is always an extensive, albeit mandatory, collection of useful subprograms. Whether or not large projects are ongoing within an organization, much reusable software is probably kept individually by many programmers. It is therefore beneficial to solicit reusable software from all of the programmers within an organization through the use of a questionnaire.

Once reusable software is selected for a library, supporting documentation must be prepared. We suggest that the documentation take two forms: internal subprogram documentation embedded within the subprogram as comments, and a short abstract describing the function of each library subroutine. All documentation should be retained in machine-readable form. If language TRGs exist, the format of the internal subprogram documentation should be their decision. The following items are considered minimal:

Use: An example of the method of invocation including a suitable set of formal arguments.

Purpose: A short explanation of the purpose of the subprogram utilizing the formal arguments of the use example.

Limitations: The assumptions which must be true at invocation so that the subprogram's correctness is assured

Warnings: Any unreasonable demands upon resources; documentation of a known, but as of yet unrepaired, error.

Required subroutines: Any subroutines which must be available to the subprogram upon invocation.

Arguments: For each formal argument given in the use example, a statement regarding its type and purpose.

Notes: General comments applicable to the subprogram or the invoking program; equipment limitations; special formulation requirements against specific arguments.

Programmer: The name and organization of the programmer who developed the original source code.

Algorithm: If the algorithm is unusual or complex, a brief description of it or a citation of its source.

Record of Modifications: For each modification made to the source code, the name of the programmer who made the change, his organization, the date and purpose of the modification.

The subprogram abstract should be a single paragraph combining most of the information contained in the subprogram comments into a terse reference. Usually, it is unnecessary to refer to much more than the purpose, limitations, warnings, and algorithm-internal subprogram comments since complete comments are readily available to the programmer who becomes interested in a particular routine.

25

The first reusable software tool is the subprogram libraries which contain the source and object codes for already identified reusable software. Both codes should be distinct but they must also be available to the programmer. The second, and relatively inexpensive, tool is an index to guide the programmer in determining the availability of a required subprogram. If considered cost-effective, an automatic retrieval system might be built by the quality assurance staff.

There is only one way in which newly developed software can be identified for possible inclusion in reusable software libraries: a policy that prohibits the development of software without prior approval by the QAP. No matter how much extra work such a procedure results in, it guarantees that an effective reusable software policy is established and enforced. Of course, mechanisms that provide for both project-wide waivers as well as an appeals process must be included. Two benefits arise from the imposition of a prohibitive software development policy:

(1) identification of truly reusable software which was inadvertently overlooked during the initial identification period;
(2) production of new reusable software to be added to the libraries.

Prior to issuing a reusable software policy, the chief of software quality assurance must establish procedures whereby his staff receives and processes requests for software development waivers. Although the quality assurance staff may be inundated by such requests at first, programmers will eventually request waivers only for software that does not appear in the reusable software library documentation. In establishing a reusable software policy, it would also be prudent to consider setting up a reusable software TRG. This TRG would assist the QAP chief in determining the criteria for accepting and rejecting proposed reusable software. Prior to the actual formulation of a reusable software standard, this TRG would additionally aid in the establishment of procedures to be followed by the quality assurance staff.

4. The COSR/SES Experiment

In December 1978, management within the Computer Oriented Scientific Research and System Engineering Support (COSR/SES) Project decided to establish a formal software QAP. The QAP's immediate goal was to increase source code maintainability. This objective was arrived at after project management discussions indicated that it would produce the greatest payoff most quickly; see Figure 3 for an idea of the quality assurance interfaces within a software development life cycle. Our discussions centered on the nature of the work perfor
within the project:

—The customer was a rese
oriented organization with progr
that required frequent modificat
—Varying degrees of quality
isted in both customer- and cont
tor-produced programs.
—Personnel employed withir
project had varying levels of skill
were organized into one- or two-
teams.
—Most project time was s
rewriting old programs to sup
new customer research. Some o
customer's programs, developed
ing the previous five to ten y
had been written with little o
concern for maintainability. M
software modifications "ripp
through the extremely com
models, making it impossible to
ercise the revised model until ea
the affected modules was ident
and repaired, one at a time.
costliness of this approach
deemed unacceptable.

Establishment of the QAP

In 1977, a relational data m
accessing method, BIODAB [1],
instituted at CSC; it resulted in
tremely low maintenance costs. A
that year, only six errors had
detected in over 17,000 line
source code; the cost to repair t
errors was less than 0.7% of the p
ect's total man-hour cost. CO
SES management decided to us

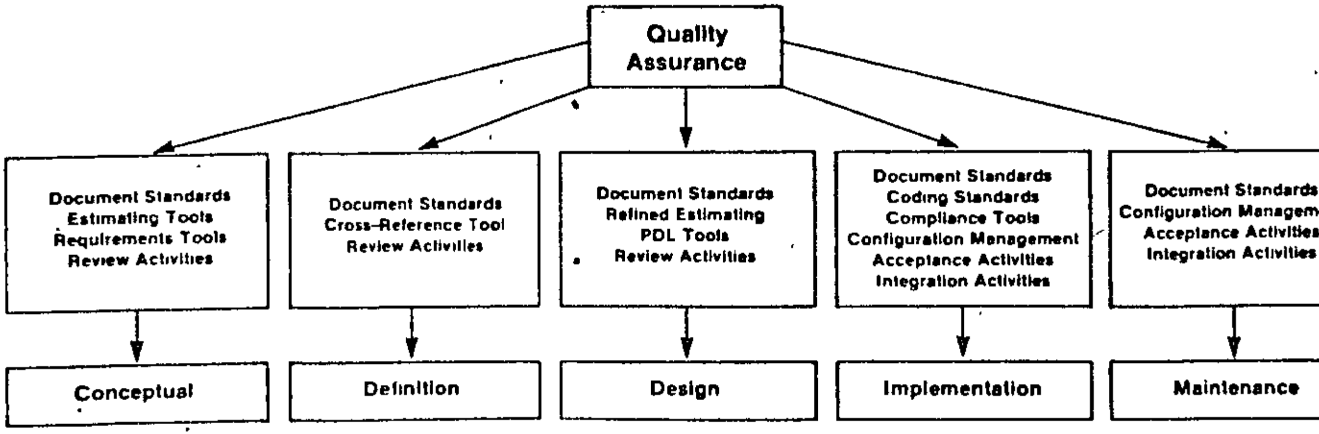


Fig 3. Quality Assurance Interfaces Within a Software Development Life Cycle.

lessons learned during the BIODAB project in developing its new QAP. The BIODAB chief programmer was invited to assume the position of QAP chief. It soon became apparent that establishing the QAP was a large enough task to require additional personnel. However, the nature of our contract would not permit additional direct labor costs to be apportioned to overhead, and so the QAP chief alone made up the entire quality assurance staff. Unknown at the time, this was a serious error!

The QAP chief's first task was writing a charter. In final form, it was perhaps overly optimistic, although essentially reasonable and acceptable in philosophy. The goal of reducing maintenance costs drove the QAP project into the last phase of software development (coding). It was felt that although a great deal of software already existed, the project could not realistically expect to contend with a total rewrite of hundreds of thousands of lines of source code (mostly UNIVAC Fortran V but with throwbacks to the older Fortran II). It was also unlikely that the customer would approve high rewrite expenditures Accordingly, the QAP plan focused on new software development with the hope that productivity as well as maintainability would, in turn, increase. In February 1979, the COSR/SES QAP Project was officially implemented. The QAP chief was to report directly to the COSR/SES project manager. All quality assurance-related programs were reassigned under the QAP.

Coding Standards Development Efforts

Once implemented, QAP initially sought to propose the coding standards for those languages most commonly used within the project (Fortran, Pascal, and Assembly). The first language chosen for standardization was Fortran. Although a Fortran 77 compiler was available, most of the project code was written in the older UNIVAC Fortran V dialect This caused the QAP chief to look very closely at that dialect and wholly ignore the other Fortran dialects in use throughout the project.

The Fortran language coding standard was intended to control the formatting of the source code and, to whatever extent was possible, to reduce program complexity. As a result, the following areas were addressed:

—single entry, single exit, and single function;
—naming and typing conventions for the source code repositories;
—length of programs, subprograms, and modules,
—label values and their positioning and attachment;
—"standard" preamble comments;
—explicit variable typing, specific ordering of type statements, and unambiguous variable names;
—statement continuation and construct indentation rules;
—making COMMON use somewhat more difficult;
—prohibiting ASSIGNS, ASSIGN'ed GOTOS, mixed mode expressions, and duplicate or null branch arithmetic IF statements.

The coding standard was then reviewed by the COSR/SES technical staff (eventually the Fortran TRG) and project managers (eventually the MRG). To provide the project's technical members with an opportunity for comment, copies of the proposed coding standards were distributed to them prior to adoption. The coding standards were eventually approved in May 1979. It soon became apparent that the coding standards were deficient in addressing the large number of dialects of Fortran used within the project. Three new members were added to the Fortran TRG and each member was asked to provide one or two dialect-specific coding standards for those dialects in which he was expert. After one year of almost constant rewriting, the new Fortran coding standards became effective in April 1980.

Our experience with the Fortran coding standards indicated that the development of the coding standard for Pascal should proceed very carefully Although the original plan was to develop a set of universal Pascal coding standards, it became obvious that Pascal implementations differed sufficiently, enough so to warrant individual coding standards for each dialect. A Pascal TRG completed its work in the summer of 1979 but the coding standard suffered from the same deficiencies as did those for Fortran. Since the COSR/SES project only used Pascal on UNIVAC equipment and all Pascal programs were written in "Navy Pascal," the general coding standard was rewritten and limited to that dialect The Pascal coding standard became effective in April 1980. Finally, Assembly coding standards for the UNIVAC-1100 Assembler were completed. As with the two preceding coding standards, the Assembly standards stress maintainability through readability.

Review of Existing Standards

Although the COSR/SES project's customers were primarily involved in research and development thereby eliminating the majority of software standards development efforts within the government, a portion of our software fell directly under military standards. However only the standards which potentially affected contract work came under the scrutiny of the QAP. Generally problems could be resolved by contacting the standard issuing organization and describing our concerns (In one case, such a call was partially responsible for the eventual replacement of an offending standard by one which is far more stringent in its requirements and up-to-date in its approach.)

Technical Audits of Existing Systems

Since our QAP chief had been involved in complex systems development, it was proposed that he oversee a technical audit of an extremely complex and costly software model. A plan was approved which provided for an audit that could be completed within 15 weeks by one, highly competent scientific programmer. Because neither the QAP nor the programmer had any experience with the system prior to the audit, we believed that the agreed upon approach would result in an objective evalution of the system.

The first order of business was talking with end-users and the sole programmer responsible for the system's maintenance. These discussions soon indicated that the system had grown beyond control. As with many simulation models, the ripple effects of a relatively minor change were magnified by the indiscriminate use of Fortran COMMON and COMMON EQUIVALENCEing. Small changes required to perform a particular new function would result in a whole system's going down. One program within the program suite was chosen for detailed study. It was estimated that,, due to convoluted code, it would take almost six months to perform a complete audit (including full comprehension of the code). The customer could not afford this kind of in-depth study and settled for a simplified report on the module.

The final report included a number of recommendations that were mostly cosmetic in nature. When these recommendations were effected, the cost of executing the model was reduced by 60% and the execution time slashed by 77% As a result of these significant savings, the customer is now considering a redesign of its system. Furthermore, the customer purchased a software configuration management tool to assure that uncontrolled modification is not repeated in the future This indicates how a QAP, although merely monitoring a technical audit, can generate revenue

Reusable Software Libraries

Prior to QAP's inception, the BIODAB project had collected useful subroutines from a number of diverse sources By the time our QAP was actually organized, four libraries had been developed However, acquisition of new routines was not formalized and maintenance was only performed as time permitted. Additionally, COSR/SES project management was initially reluctant to support continued library development (a reluctance which was justified, in part, by the high costs of subroutine documentation and maintenance). Due to one project's slow start-up, though, two programmers became available as temporary quality assurance staff members. Their skills were applied to the documentation of a Fortran V subroutine library consisting of 45 routines. Within two man-months, this documentation was complete, and currently, the Fortran V subroutine library is being referenced by a number of projects on the average of 50 times per week. (The library which sees the highest use, an estimated 500 accesses per week, is an editor procedure library that was developed to support the University of Maryland text editor user. Its documentation was also completed as time allowed.)

Performing maintenance only as time permits results in serious deficiencies. One project, operating under very tight time constraints, required the services of the Fortran V library A routine of particular importance to the project was found to be in error (it had been for over one year). Unfortunately, without a reusable software policy, no notice was given to the QAP. The result was that four days of potential productivity were lost as the routine was rewritten Had a reusable software policy been in effect, the programmer who discovered the error a year before would have been forced to report it Now apparent, due to the success of the two documented libraries, is the fact that reusable software can significantly increase programmer productivity As a result, the COSR/SES QAP has implemented a full-scale documentation and maintenance project for the two remaining libraries (Pascal and UNIVAC Fortran 77), as well as developing a new Fortran library for the DEC VAX-11/780 Upon completion of these tasks, a formal,
much-needed reusable so
icy can be implemented

Software Development Consult

One facet of the QAP pr
plan never considered during
velopment was the adverse
that would be felt if the QAP
in addition to performing its q
assurance duties, had to advise
projects. Almost daily, the QA
was consulted on various asp
software development and eng
ing. The result was a series o
cational programs, consul
with managers of projects tha
in serious difficulty, and occas
temporary ressignments to p
requiring the immediate ex
and guidance of a senior pro
mer. Providing these services
the QAP to actually forego p
tive quality assurance develop

5. Conclusions

The effectiveness of a s
QAP is directly related to its f
and staffing support and the t
with which management purs
program's objectives If a s
QAP is going to be truly effe
must be staffed with a su
number of technically co
personnel Once staffed, th
should not be disrupted by
temporary reassignment of it
bers

In light of our experienc
the COSR/SES Project, we
that the hallmark of a su
QAP is the ease with which s
products, produced employ
tenets, can be maintained. A
there may be a relatively stee
learning curve, the producti
organization with an est
QAP will gradually surpass
an organization lacking one

Acknowledgments

We wish to thank our co
G S Key and T.A Johnson
comments on this manuscr
evolved We would also lik
press our appreciation to th
bers of the COSR/SES Pr
their forebearance as the
periment proceeded

INSIDE.
Structured Walkthroughs – Concept and Purpose, Mechanics of the Method, Problems with Structured Walkthroughs, Management and Organizational Support and Participation

28

Structured Walkthroughs

PAYOFF IDEA. Detecting software errors and modifying programs are difficult tasks because of the way in which programs are constructed. Tools, techniques, and guidelines that address these problems are being developed. One such technique is the use of structured walkthroughs to "proof" program design, detect errors, and control structure. There is evidence that using structured walkthroughs can improve programmer productivity This portfolio discusses the concept of structured walkthroughs, how this technique can be applied, and what it can do for your organization

PROBLEMS ADDRESSED

Before starting to construct any system, program, or module, it is advisable to ensure that the construction will be suitable. This is the purpose of a structured walkthrough A structured walkthrough is simply a review of a system or a software product by people involved in or allied with the development effort, in other words, people at the same level in the hierarchy review a development effort together to find areas where improvements should be made.

Typically, structured walkthroughs are associated with programming and software construction. Such walkthroughs are aimed at uncovering errors in code But this is only one way to use walkthroughs; they are also useful for design reviews: A team can review such design particulars as file types, access methods, data base design, planned coding schemes, and so on. Similar reviews can be undertaken at the requirements analysis stage so that any omissions, misunderstandings, poor decisions, or vague areas can be given attention before additional time and resources are committed to the effort.

The Need for Structured Review. Software systems are the lifeblood of the modern computer system. It is no secret that although hardware

23

costs are decreasing rapidly, software-related expenses are rising at a rate that is totally out of control Furthermore, even with the high cost of building or acquiring programs, it is difficult to obtain high-quality software Even purchased software suffers from lack of quality. For example, the IBM OS operating system contained many errors; this was true even in later versions, which had been used at hundreds of field sites. This problem is not uncommon.

Although the industry has not yet devised a way to estimate software construction time accurately, it is clear that development time is often excessive. A goal of virtually all organizations is to obtain high-quality software more quickly, especially if this can be done at a lower cost than that currently being incurred. Structured walkthroughs can help achieve this goal

After software systems are built and installed, they are often used for many years, during which time they undergo changes. Some changes correct errors; others add new functions or enhancements. In addition, maintenance is performed to rewrite sections of code so that a program is more efficient (see Table 1).

Unfortunately, even experienced programmers often do not write software that is easily maintained. Possible modifications are not considered during the design and programming activities; thus, ease of maintenance is not designed into the code. By calling attention to this area of activity during design reviews, walkthroughs can aid in formulating design methods and coding standards that make program maintenance and enhancement easier. In addition, because proper use of walkthroughs can reduce the number of programming errors, the need for corrective maintenance can be reduced. Each of these advantages will become more apparent when the discussion examines ways to conduct software reviews

The Purpose of Structured Walkthroughs. When conducting a walkthrough, the goal is simple. to find errors or problems. Note that no attempt is made to correct these difficulties at the time they are found For reasons that will become more evident later, this is done sometime after the review is concluded

Table 1. Maintenance Activities Associated with Software Systems

Maintenance Type	Description
Repair and Correction	To correct operating deficiencies, errors and bugs that have been detected or are known to exist in the software or processing system or in the overall application design
Revision	To install mandated changes due to business or environmental (e g , government) inputs that require modification of the system, also includes changes to improve job or program efficiency
Enhancements	Modifying an existing system or application to do additional processing or reporting or to do additional functions

It is also important to emphasize that structured walkthroughs are not intended to find fault with or blame any individuals. The design or the software, not the designers or programmers, is the focal point of the review. The emphasis of the discussion must be on improving the product, not on assessing individuals. If this strategy is violated or abandoned, structured walkthroughs lose their meaning and value. In addition, involved personnel will begin to consider a required walkthrough an obstacle to be avoided rather than a helpful, cost-saving tool.

Relation to Training Programs. Training for people who are new to the department or new to a certain job is often neglected. Many organizations rely on some outside organization (e.g., a college or university or another business) or on attendance at professional seminars or workshops for training Although each is useful, they are not adequate because actual in-house methods and situations cannot be addressed.

A common way of introducing new programmers to a job environment is to put them on a large-scale software maintenance project and have them clean up or adapt a software system to meet stated operation and quality specifications. Such projects are usually one to six months in duration. The rationale is that by correcting and/or enhancing someone else's work, they can obtain skills and techniques while learning about common mistakes and how to avoid them

Participating in structured walkthroughs extends such training methods. Walkthroughs can focus on live, in-house development or maintenance projects, at various stages of development, such as specification, design, and programming. Walkthroughs not only focus on problems and errors, but also can show ways to detect and avoid these kinds of difficulties or inefficiencies.

Structured walkthroughs also offer training in a different sense. Each member of a walkthrough team must be an active participant, not a casual observer. While making contributions, they learn methods and gain insights from other people on the team. Thus, both junior and seasoned programmers learn and advance.

CONDUCTING STRUCTURED WALKTHROUGHS

The manner in which structured walkthroughs are conducted can largely determine their usefulness. In this section, the way to get the most out of this technique is discussed. The focus is on when to use the method, how to select appropriate participants, what procedures to follow, and what the result should be.

When To Do Structured Walkthroughs. Because walkthroughs are aimed at producing more reliable and cost-effective software products, there are several points in the software life cycle where walkthroughs can be applied: after the analysis stage, following design, and after programming. These reviews seem to be of equal value (see Table 2).

Requirements Review

Following investigation and determination of the information and processing requirements that the proposed design should satisfy, it is

Table 2. Types of Reviews Where Walkthrough Methodology Can Be Used

Type of Review	Description
Requirements Review (Specification Review)	Performed after a preliminary systems investigation has been completed in order to determine which functions and activities can or should be handled by the proposed application system Also aimed at identifying misunderstandings, inaccuracies in specifications, or misleading assumptions on the part of either systems personnel or users
Design Review	Performed to examine the logical design of the applications software Assessment of the system blueprint as it has been established to determine whether the design will meet the original design specifications
Code Review	Aimed at detecting problems in the coded software stemming from errors, misunderstandings, or poor adherence to programming standards The review should ensure that the code meets the original design specifications (Applies to both new developments and maintenance projects)
Testing Reviews	Performed to ensure that the testing strategy being used for an application is sufficient to detect the most significant errors or bugs in an application program Includes review of test data to be used

often useful to walk through the requirements specifications. This review, which is called either a requirements review or a specification review, is directed at examining the functions, activities, and processes to be handled by the forthcoming system. Any inconsistencies in the requirements stated by the users or identified by the analysts should be uncovered, as should any areas in the specifications that are vague or unclear. Inaccurate statements and assumptions should also be detected.

It is generally suggested that a requirements review focus on a written document that can be studied prior to the review session. Narrative descriptions that explain the context of the system are commonly used for requirements walkthroughs. These descriptions should spell out the different activities in the system area under investigation. Narrative descriptions should also identify key people and components and how they relate to one another. Sources and uses of information should also be identified.

Some shops use flowcharts that relate processes, control points, and data flow, or they use decision tables or even data dictionaries in place of narratives. The particular form of the description is unimportant as long as it is useful and understandable to those involved in the requirements review.

Design Review

As its name implies, a design review is a structured walkthrough to examine the logical design selected to deal with the information and processing requirements that were identified in the systems analysis stage. This review attempts to determine whether the proposed design is a valid one and if it will meet the specifications. The walkthrough can be conducted by examining a design presented using any one of a number of documentation/presentation methods such as HIPO and pseudo code.

Code Reviews

The type of review that is usually associated with structured walkthroughs is the code review. Most organizations and staff members begin using the walkthrough methodology with this form of review. Quite simply, a code walkthrough is an assessment of program code.

For new development projects, participants may review the entire software set as a complete package, comparing it to the original design or requirements specifications. Discovering that a portion of the code does not agree with the original specifications or finding problems or mistakes that originated with the earlier requirements analysis is not uncommon. Although having to modify the design or change the code at this point can be frustrating, it is easier and less costly than waiting until the software is installed.

As indicated earlier, code reviews should not be limited to new development projects; maintenance activities can also benefit from structured walkthroughs. The method is the same and the benefits equally significant. Maintenance projects have all the attributes and problems of a new development activity, except that in maintenance projects some of the code and part of the documentation is already completed when the work starts (note that this can be an advantage or a disadvantage).

Testing Reviews

Many organizations overlook the advantages of using the walkthrough methodology to review testing, yet the same benefits of peer review can be realized at this life cycle stage also In fact, errors undetected during the testing stage will likely remain with the system when it is implemented, and it is these errors that cause nightmares for users and programmers alike.

Participants in testing reviews do not actually examine the output from test runs or search for errors that have been detected using a set of test data. Rather, they focus on the testing strategy to be used and determine whether it is adequate to catch critical errors. These people also assist in developing test data that can detect design or software errors. The purpose of testing is to find errors, not to prove program correctness; therefore, an effective testing strategy is one that is likely to find the most serious errors.

THE PARTICIPANTS

One of the first questions to arise with structured walkthrough methodology concerns selection of participants. This issue and how it is handled can significantly affect the usefulness of the review strategy

The Role of Programmers/Designers

Although programming and systems development have generally been considered solitary activities, this notion is changing. The concept of programming teams has become a topic of frequent discussion in many organizations. Through the team approach, more timely and reliable code is expected.

The use of structured walkthroughs is related to team programming, not just because they are based on similar objectives, but also because the walkthrough concept recognizes that systems and program development may involve several people for each step. Structured walkthroughs, then, are team-like activities. Furthermore, participants, with the possible exception of representatives from user departments, are individuals who are actually involved in developing software or applications.

The individual(s) who formulated the design specifications or wrote the code being reviewed, as well as a number of other people, in the walkthrough should participate. Often a moderator is chosen to oversee the walkthrough and keep the group focused on the discussion topic, (e.g., finding, not correcting, errors and problems). The moderator does not necessarily have to lead the review; many organizations prefer to have the programmer or developers do this because they are the people most familiar with the details of their project. (On the other hand, this familiarity can be a problem by introducing strong biases or persuasive capabilities and causing reviewers to overlook problems inadvertently.)

It is imperative that the information produced during review sessions be captured completely and accurately. The leader of the session is occupied with ensuring that the appropriate concerns are discussed, and therefore he or she may not be able to jot down all of the points aired by participants. The programmer or developer may not record ideas in the same manner in which they are discussed by reviewers. Therefore, it is advisable to appoint a recorder for each walkthrough session, in order to have all relevant details recorded completely and objectively. The intention is not, of course, to get a highly detailed record of who said what, but rather to record the important points made. During the review, comments and suggestions may be made in rapid succession. Thus, the recorder must be constantly attentive. In many sessions, recorders are so busy taking notes that they cannot participate.

Experienced data processing organizations are finding that standards for data names, module determination, field type and size, and so on are desirable This is most often discussed in relation to data base environments, although it is equally important in non-data-base environments. In any case, the time to start enforcing these standards is at the design stage. They therefore can be discussed during the walkthrough sessions. This discussion can be led by the moderator or by a representative of the standards or data administration group.

Maintenance considerations should also be addressed during the structured walkthroughs, such concerns include coding standards, modularity, documentation, and parameterization. It is increasingly common to find organizations that will not accept new software for installation until it has been approved by software maintenance teams. In such an organization, a maintenance representative should be included in the review team.

Role of Management

Generally, management should not play a direct part in a walkthrough. Doing so would jeopardize the intent of the review. Reviews are aimed at helping individuals improve their design or product and, at the same time, creating a cost-effective software product for the organization. This discussion has characterized walkthroughs as occurring in an open, give-and-take atmosphere. Unfortunately, if management takes an active role in walkthroughs, it is likely that the true spirit of the review will dissolve

Too often, management involvement is taken to mean evaluation. Many times, the result is that individuals attempt to perfect their product before the review session so that they look good in the eyes of management. Managers may feel that a high number of questions, mistakes, or changes indicates that the individual whose work is under review is incompetent. In brief, when management attends walkthrough sessions, the atmosphere changes significantly, resulting in less constructive results for the organization and the other participants.

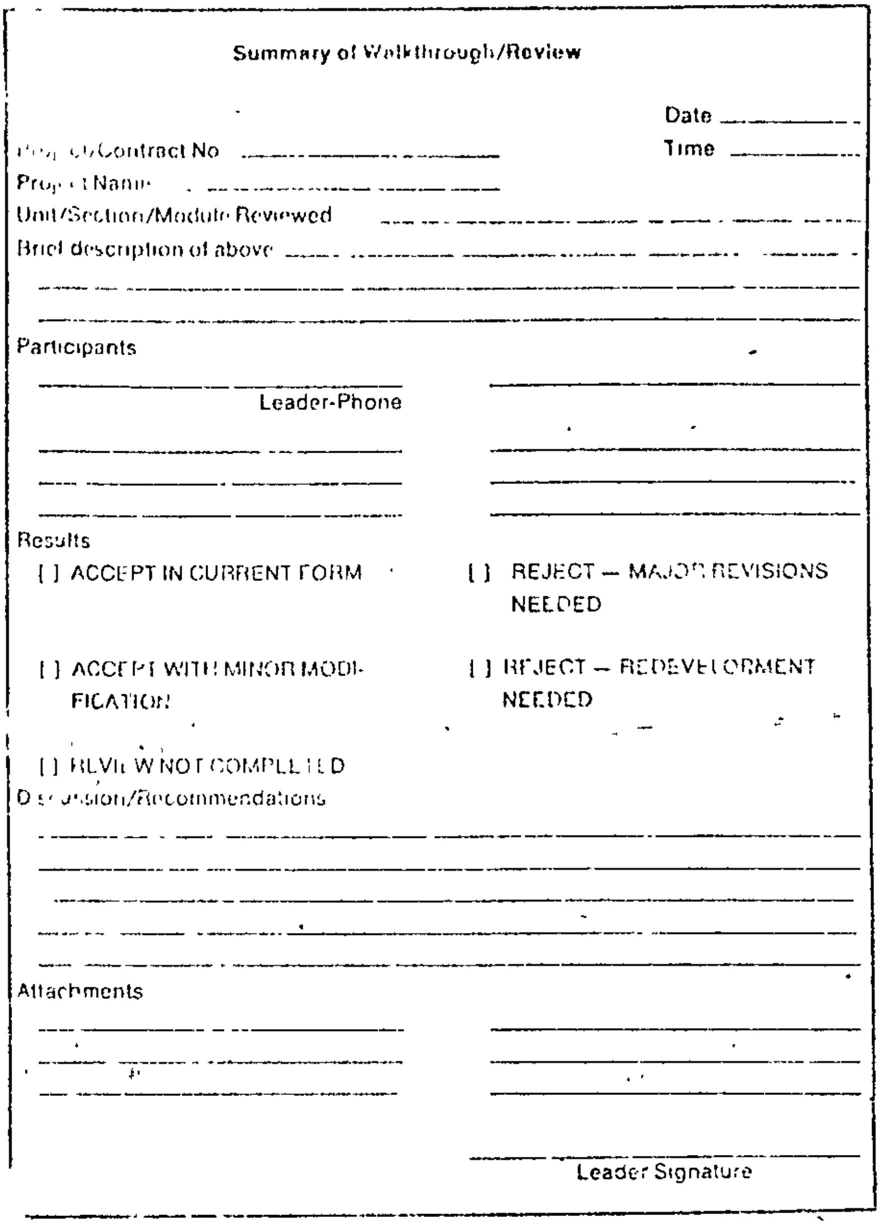
Management may ask for reports summarizing the review sessions. Some types of reports, however, should not be produced. The only information that really need be passed on to management is that a review has been done, which project or product was discussed, and who attended or participated. Reports should not summarize the errors detected, modifications suggested or revisions needed. If participants know this information is communicated, it will have the same effect as management actually observing. An appropriate sample evaluation summary is shown in Figure 1. This could be augmented with a one- or two-page memo giving a bit more detail.

Although it may seem unrealistic that management should not be involved in structured walkthroughs, most managers indicate that they prefer not to attend these sessions. They recognize that the walkthrough is a work session rather than a time to evaluate staff members. They also realize that because the sessions can be quite technical and clearly require a detailed knowledge of the product being reviewed, they would not be able to contribute much to the discussion. Moreover, managers are usually aware that their attendance can change the atmosphere of the sessions, thus inhibiting progress.

Size of Walkthrough Team

The members of the walkthrough team should be carefully selected so that the various roles are filled by competent and contributing people. Care must also be taken to ensure that the size of the team is appropriate for the project under review. At a minimum, the team should include the individual(s) who actually designed or coded the project, a recorder, and a leader.

In some organizations it is felt that having more persons involved in the examination increases the chances of locating problem areas. The group should not be so large, however, that lengthy discussions are needed, review sessions should not exceed 90 minutes. Considering the time constraint and general purpose of the review sessions, it is suggested that an upper bound of about seven persons be set for any walkthrough



Summary of Walkthrough/Review

Date ____________

Project/Contract No ____________ Time ____________

Project Name ____________

Unit/Section/Module Reviewed ____________

Brief description of above ____________

____

____

Participants

____________ ____________

Leader-Phone

____________ ____________

____________ ____________

____________ ____________

Results

□ ACCEPT IN CURRENT FORM

□ REJECT — MAJOR REVISIONS NEEDED

□ ACCEPT WITH MINOR MODIFICATION

□ REJECT — REDEVELOPMENT NEEDED

□ REVIEW NOT COMPLETED

Discussion/Recommendations

____

____

____

____

____

Attachments

____________ ____________

____________ ____________

____________ ____________

____

Leader Signature

Figure 1. Sample Form for Reporting Results of Walkthrough Session

Organizational Support and Participation

Although DP management should not have any direct role in walkthrough sessions, users should participate in such nontechnical walkthroughs as those conducted to examine specifications or functional requirements. Users can be extremely helpful in recognizing problems in system design attributes.

Some users may criticize walkthroughs as too time-consuming for the results they produce. This occurs when the users do not fully understand the purpose and method of reviews or because they have had poor experience with structured reviews. In these cases the problem is not the review method, but rather the way it has been implemented. In general, when structured walkthroughs are properly introduced and administered, the results are apparent to systems persons and users alike because more timely and correct systems are obtained.

PROCEDURES

Walkthroughs depend on fully informed participants, thus, those involved must come prepared. The individual requesting the walkthrough (i.e., the designer or programmer whose work is to be reviewed) should notify participants far enough in advance that they can study the materials to be examined. Generally, two to three days notice is adequate.

Which materials should be distributed depends on the type of walkthrough to be conducted. Copies of the documents or code to be walked through should be distributed, along with summaries of interviews, sample forms, and so on for a requirements review or system descriptions, input/output charts, and macro flowcharts for a design review. Code and test reviews usually require program listings and test data plans.

It has already been pointed out that walkthroughs should not be too lengthy. Because time and concentration limits preclude a single-session review of, for example, a 10,000-line COBOL program, it is obvious that this amount of code should not be distributed. Rather, only the modules actually being examined should be distributed. If this cannot be done easily, it may be an indication that the design is not sufficiently modularized.

It is essential that the participants in a walkthrough have the time, interest, and willingness to do the required preparation. If they cannot or will not prepare adequately, they will not be able to contribute to the walkthrough It is better to have others replace them on the team so that maximum benefit can be realized. Note that if participants are expected to spend five or six hours in preparation, they may be justified in claiming they are too busy In this case, it is probable that the session is not organized or limited properly. The objective of the session must be reformulated.

Starting the Walkthrough

There is more than one way to handle the mechanics of the walkthrough; the best approach depends on the organization, the nature of the people on the review team, and perhaps even the type of project being examined. It is generally suggested that the moderator for the session, rather than the programmer or designer, start the session and introduce the plan of action. The moderator may prefer to have the programmer or designer then give an overview of the project, presenting the important attributes of the design, code, and so on. It is recommended that the moderator ensure that this be an objective presentation that tells the what and how of the segment to be reviewed rather than the why. The presentation should not be a defense given before the wolves start in with their cross-examination. If the review involves a project with which the participants are already familiar from previous walkthroughs, it may be unnecessary to begin with a formal introduction or overview.

How to Proceed

Depending on the type of review and whether or not it is the initial one, the actual walkthrough activities may vary For first reviews of a project, attention should be given to determining what the logic or specifications are in comparison to what they were intended to be. For example, if processing logic does not perform all required validation checks, the team should discover this. As difficulties or misunderstandings are uncovered, they should be noted by the recorder so that they can be dealt with later.

If design specifications are being reviewed, participants should consider whether the proposed design will do the intended job and if it will do it efficiently. Answering these questions necessitates knowing about such items as file and transaction volumes, update frequencies, processing modes, access methods, keys, and the like. In addition, participants must know something about how the output from the system will actually be used; they should also be informed about the type of people who will use the system so that interface methods and protocols can be scrutinized.

When programs are reviewed, the participants also have to be sensitive to execution efficiency, use of standard data names and modules, and program bugs. Appropriate comments and documentation permit this level of scrutiny. Obvious errors, such as syntax errors and blatant logic errors can even be jotted down ahead of time by team members and submitted to the recorder, thereby saving meeting time. Other errors may merit discussion and examination during the review. Figure 2 shows a section of a checklist that might be used for noting problems and their severity

If the review session is not the first one, there is "old business" to handle; that is, problems, suggestions, and comments mentioned during the previous walkthrough must be resolved. The designer or programmer should indicate how problems were solved, which suggestions were taken, and which were not taken, along with the reason(s) for choosing alternative solutions. There may be very good reasons for not making changes as suggested, but they must be communicated to and agreed upon by other participants. When all of the old business is cleared, other areas may be reviewed.

Approval

In many organizations, the team assigned to do structured walkthroughs on a particular project has final approval authority on the project. In other words, the team must approve the specifications, design, code, and test plan before the project can proceed to the next stage. In some cases, a project with problems that are corrected need not be

Walkthrough/Review Checklist

Project/Contract No ____________________

Project Name ____________________

Review Category	Problem Detected (check all that apply) Absent	Unnec	Error	Major*
Backup procedures				
Error messages				
Execution time				
External documentation				
Internal documentation				
Input validation				
Interface mechanism				
Procedural logic				
Passing of data				
Meets design specifications				
Meets user/problem specifications				
Meets coding standards				
Meets data standards				
Maintainability				
Storage use				
Test data				
Test procedure				
Test for end conditions				
Test for all possible conditions				
Test drop through				
Transfer of control				
Visible structure				

* Major error that will cause failure or crash

Figure 2. Sample Checklist for Guiding Review Activities

returned to the team for a final review prior to its acceptance When critical changes are involved, the team may decide that review of the modified program or design is required before acceptance is granted.

Some organizations have set up formal voting rules stating that unanimous agreement must be reached for acceptance. In others, a simple majority is necessary. In still others, the formality depends on the nature of the problem or change suggested; design questions may require full team approval while questions of programming taste or execution efficiency may be settled with the approval of only a majority of team members.

PROBLEMS AND PITFALLS

As previously noted, problems can develop if management becomes directly involved in walkthrough work sessions or if participants do not adequately prepare for reviews. There are, however, other ways in which problems can develop.

If there is a tendency to try to complete a walkthrough too quickly (perhaps because the participants have some other meeting or project that is demanding their time), the walkthrough will not be fully successful; in the rush, problems or weaknesses in the product can easily be overlooked. It is generally better to postpone a walkthrough until the participants are able to devote the necessary time to do it.

Some walkthrough teams get enmeshed in discussions of programming style. Although adherence to organization standards for such items as data names, field length, and type is important, inflexibility with regard to style can be counterproductive, especially if it is based on personal preference. Avoiding this type of difficulty is the task of the leader or moderator.

A similar concern regards individuals who feel they have *the* answer to a particular problem or situation. One of the objectives of structured walkthroughs is to set ego problems aside; however, an individual pushing a single correct approach is bound to occur. The participants and leader are responsible for recognizing and subduing such a person, even if it means openly ignoring his or her suggestions.

Egos are manifested in other ways also. For example, some people enjoy making others look bad. They therefore attempt to find "just one more error" or discuss an approach as being "the worst way to do it." Unfortunately, these problems are all too common.

Some pitfalls and problems are more subtle than the ones previously mentioned. The individual who is always late or the person who is in too many walkthroughs cannot do justice to any of them; such a person needs to be better managed. Similarly, participants who fake it by making a leader or developer render extra detail as a way of covering up their unpreparedness do not contribute to the team. Those who are not willing or able to face the real problems with a design, for example, and attempt to sidestep such problems by blaming the situation on standards or users represent a different kind of problem.

RECOMMENDED COURSE OF ACTION

Merely having a structured walkthrough for a program or project does not guarantee that the final product will be better than if no walkthrough had been held; the approach is an effective one only if used properly. To ensure success, the walkthrough must be properly managed and conducted in accordance with the guidelines outlined in this portfolio.

This portfolio was written by James A. Senn, Associate Professor of MIS at the School of Management, State University of New York, Binghamton.

Reference

Fagan, M E ' Design and Code Inspections to Reduce Errors in Program Development '' *IBM Systems Journal*, Vol. 15, No 3, 1976, pp. 182-211

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

PLANEACION DEL PROYECTO
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

iv. PLANEACION DEL PROYECTO
(DOCUMENTACION COMPLEMENTARIA)

C A P I T U L O I

¿ Qué es la *Planeación* ?

1.1 La *Planeación y* sus conceptos fundamentales :

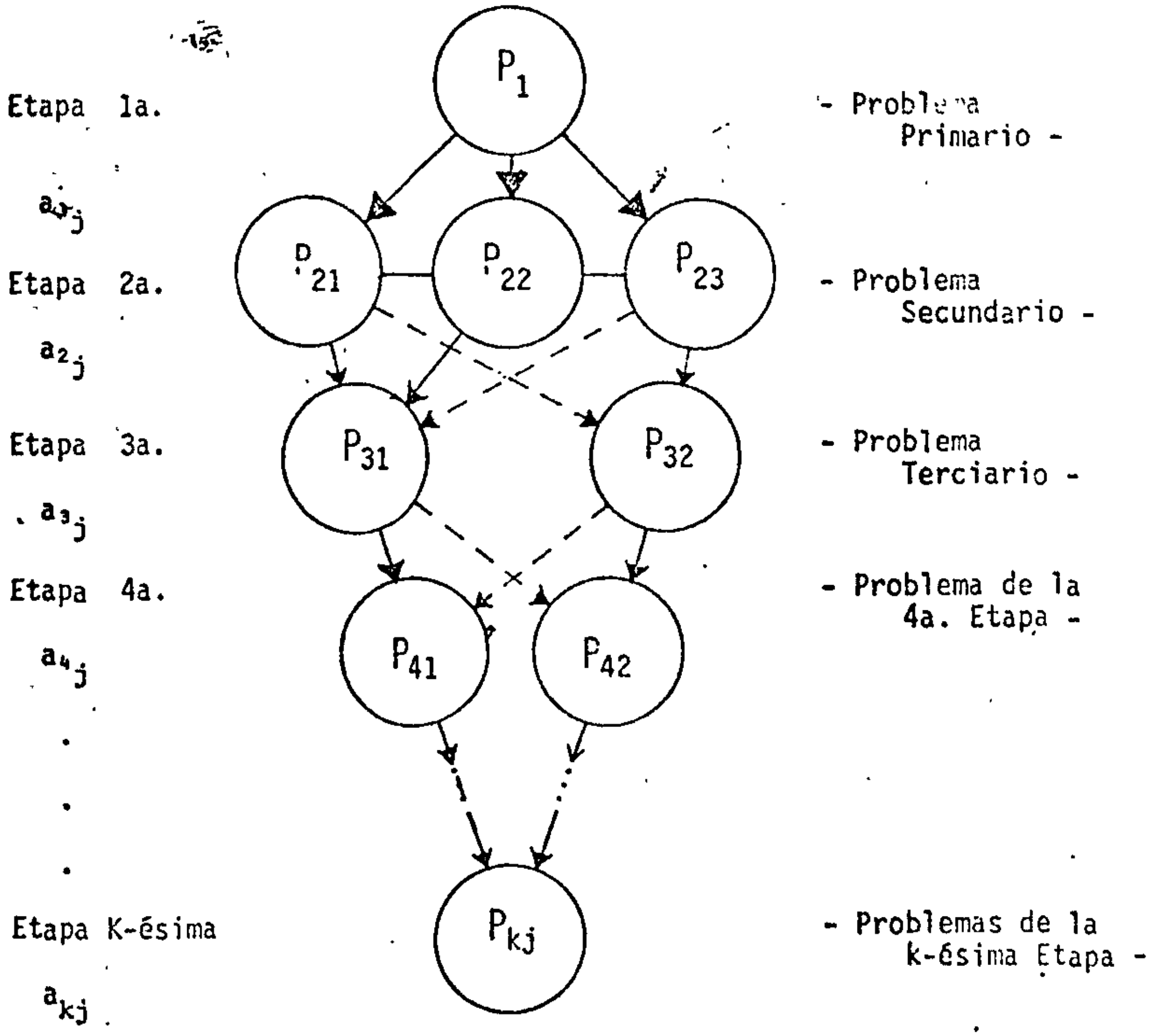
Empezaré por situar, en base a una primera definición de la *Planeación*, -- el contexto general de discusión : una definición de *Planeación*, nos ---- dice que " *es un proceso de Toma de Decisiones antes de que la acción - se realice, en donde las alternativas para alcanzar el objetivo son varias e interdependientes* " (44). Al ser un proceso de Toma de Decisiones debe estar enmarcado en una situación problemática compleja, en -- donde se detectan un conjunto de problemas interdependientes (una malla de problemas) cuyo objetivo es resolverse en forma global de acuerdo a - los objetivos propuestos. En ese sentido, es un *proceso* (1) - en términos más estrictos - pues existe una diacronización de problemas en la - malla y se define *etapas y estados* de la resolución. Más aún, es un proceso de evaluación y orientación constante que debe considerar -necesariamente - la implementación (y los efectos de ésta) de cada alternativa. Dicha primera definición aunque incompleta nos provee de elementos generales de discusión que se pueden reflejar en el siguiente diagrama :

- DIAGRAMA 1 -

- MALLA ESTRUCTURADA DE PROBLEMAS -

Donde cualquier alternativa global de solución debe considerar la red --- (malla) de problemas. Es decir,

$$A_h = (a_{h_1} , a_{h_2} , a_{h_3}, \dots , a_{hk})$$



Como se observa en el diagrama anterior se ha considerado un modelo de --- decisión estructurado - lo cual no siempre sucede en la práctica, aunque con algunos supuestos adicionales se puede llegar a él - (2). De todas formas, exhibe el resultado de que en todo proceso existe una sucesión de acciones (por lo que en todo proceso de decisión se toman decisiones sucesivas) y que todo proceso de decisión es *Teleológico* (3) y así, se puede obtener una analogía entre la *planeación* y el *diseño* si el proceso de Toma de Decisiones es un proceso que satisfaga el contener :

(i) un componente indicador-descriptivo.

(ii) un componente de ajuste entre las alternativas sucesivas y el conjunto de metas especificadas (de Evaluación)

(iii) un componente de comunicación (que cubra los problemas de no tener un código común y las dificultades de implementación que se deduce de ello, etc.)

(iv) un aspecto estructural general - en el sentido de que exista una Metodología común, pero que sea flexible, pues se considera que todo problema es en sí mismo -- distinto. En otras palabras, que la *Teoría Filosófica* de apoyo y la metodología de estudio sea común, aunque cada problema se pueda visualizar en sus particularidades propias.

Ahora bien, si la *planeación* es *Teleológica*, ¿ cuáles serán sus objetivos ? Una definición (1) nos indica que la *planeación* es " *tener la mayor certeza posible de lo que suceda en el futuro, de acuerdo a los objetivos establecidos y no de cualquier otra manera* " . Esa última definición es criticable, pues considera ajeno a la *planeación* el establecer los objetivos en el Sistema o en la Organización (4), que debe - en mi opinión ser una de las actividades del planeador (sobre todo en su participación con la comunidad en estudio). Por òtro lado, el tener " *mayor certeza posible de lo que suceda en el futuro* " será un aspecto de la *planeación* - (*La Previsión*), y que confundir con el aspecto global de la misma, será un error muy costoso (por ejemplo, prever las necesidades de cierto sector en el futuro si no se toman acciones o decisiones ¿ será planeación ?) (4).

Otro autor - Ackoff - nos dice lo siguiente : " *Planear es efectuar una toma de decisiones anticipada. Dichas decisiones consideradas forman un Sistema de partes interdependientes. Ya que el Sistema es demasiado grande y complejo para manejarlo globalmente a la vez, la planeación debe realizarse en partes y cada parte debe ser evaluada a la luz de por lo menos otra parte* " (3), " *Planear es el proceso de buscar soluciones a un conjunto de problemas interdependientes* " (6), " *la planeación es proyectar un futuro deseado y los medios efectivos para conseguirlos* " (9). Así el objetivo de la *planeación* puede representarse como :

a) la búsqueda de <u>soluciones</u> (por lo tanto ha de resolver un problema complejo constituido por problemas interdependientes).

b) la de proyectar (y de <u>obtener</u>) un futuro deseado . Por lo -- que, se pueden exhibir globalmente dichos dos objetivos enunciados de la manera siguiente : La *Planeación* tiene como objetivo - el de buscar un futuro deseado en un Sistema Social y resolver - así un conjunto de problemas interdependientes.

De esa forma y coincidente con la definición mostrada (44) al indicar que " *la planeación consiste en determinar <u>lo que</u> se deba hacer, <u>cómo</u> debe hacerse, <u>qué</u> acción debe tomarse, <u>quién</u> es responsable y <u>por qué</u>* ", -- se obtienen los siguientes puntos mínimos :

1.1.1. I) La *Planeación* es una actividad *Teleológica* (relacionada con el *Diseño*) ;

II) la *Planeación* es una actividad anticipativa ;

III) La *Planeación* envuelve la resolución de problemas complejos en *Sistemas*, donde los problemas son interdependientes ; (por lo que todo tipo de decisiones está inter-- relacionado a las decisiones previas y a las futuras) ;

IV) La *Planeación* se realiza en *Sistemas Sociales* (que son abiertos y dinámicos (5)) ; por lo tanto y siguiendo ese enfoque una discusión más formal tomaría, entonces el siguiente procedimiento :

a) Especificar qué es un problema de *Planeación*.

b) Identificar qué se entiende por *Planeación*.

c) Definir qué es un *Plan*.

d) ¿ Cómo se debe *Planear* ? (El aspecto normativo de la *Planeación*).

Al proseguir la discusión, se especificarán, con mayor profundidad, algunos de esos puntos mínimos y ,otros que, mediante aportaciones de expertos o -- debido al procedimiento de discusión, sean importantes.

Si la *Planeación* está en un esquema de Toma de Decisiones, debe deducirse - evidentemente, que existe problemas o situaciones problemáticas propiamente de *Planeación*.

Para formalizar la proposición anterior, se propondrá el siguiente esquema de discusión :

1.1.2 Problemas de *Planeación* .

Las siguientes, se postula - al estilo de Ackoff - como condiciones mínimas para que exista un problema de *Planeación* :

a) Un individuo o una Organización que tiene el problema : el - *Tomador de Decisiones* (T. de D.)

b) Por lo menos, un objetivo que el T. de D. desea satisfacer.

c) Por lo menos, dos alternativas que púedan satisfacer, el --- (los) objetivo (s) .

d) Un estado de duda " a priori " de parte del T. de D. sobre cual alternativa es la mejor.

e) Un Ambiente o contexto del problema. (El ambiente consiste en todos los factores que pueden afectar la consecución del objetivo y que no están bajo el control del Tomador de Decisiones).

Es decir, mediante las condiciones anteriores, lo que se está exhibiendo -- ante todo - es que cualquier problema de *Planeación* debe ser un problema (en el sentido que define Ackoff (1)).. Pero, aún existen varios puntos adicionales que los caracterizan como un problema de *Planeación*, -- como los que se presentan a continuación :

f) Todo problema de *Planeación* está embebido en un Sistema, por lo que, los problemas deben resolverse según un enfoque Sistémico. Así, dado que los componentes del Sistema están in-- terrelacionados, las decisiones y las alternativas de solu - ción son interdependientes.

Cada grupo de alternativas de decisión tiene una diacronización, ya que se plantean en forma sucesiva, y, por lo tanto, cada sucesión de alternativas elegidas, estructura una *Estrategia de decisión*.

g) En otras palabras, se presenta una sucesión de problemas interdependientes, y para cada uno de ellos se propone distintas alternativas de solución que configuran una estrategia - general de solución (por ejemplo, ver diagrama 1)

h) Todo problema de *Planeación* pertenece - por su propia definición - a un Sistema Abierto y Dinámico. Por lo que el --- tiempo y el horizonte de *Planeación* elegido tienen una gran importancia en el planteamiento y sobre todo, en la implementación de la solución

i) Como se mencionaba ya anteriormente, el tipo de Sistema o -- Cominidad donde se aplica la *Planeación*, tiene también caracterśiticas importantes, puesto que - como se verá posteriormente - una *Taxonomía* de los problemas de *Planeación* debe - incluir el marco de aplicación.

Aún más, los problemas de *Planeación* ejemplifican las dificultades que menciona Ackoff (1), ya que, por ejemplo, existe más de un Tomador de Deciones ; los Tomadores de Decisiones son distintos de los beneficiarios ; los Tomadores de Decisones no implementan - en algunas condiciones - las estrategias elegidas y siempre existen reacciones y contracciones en el -- Sistema ; asimismo puede considerarse no sólo un objetivo sino un conjunto de ellos que no necesariamente es consistente.

En el anterior planteamiento, se han mencionado algunas palabras básicas, algunas de ellas se aclaran en las notas al final de la tesis y, otras, por conveniencia, se definirán a continuación.

Alternativas : Son cursos de acción que puede elegir el T. de D. para cada problema interdependiente.

Estrategias : Es el conjunto de alternativas que se eligen, considerando, hasta cierto instante de tiempo, a una colección de problemas interdependientes.

Políticas : Se refiere a las estrategias de solución en situaciones estáticas.

Planes : Un plan identificará la selección de estrategias en casos dinámicos.

Podría parecer fuera de contexto esas dos últimas definiciones, si el lector recuerda lo ya mencionado anteriormente : " *que todo problema de Planeación está en un Sistema Abierto y Dinámico* " , luego, siempre se tendrían Planes y nunca Políticas. Pero, más bien, la distinción entre Plan y Política obedece a la forma como se visualiza el problema y a los elementos en el horizonte de Planeación que se consideran. Por ejemplo, una *Planeación* a corto plazo (*11*), que considere al Sistema como estable y casi estacionario, estaría utilizando Políticas y no Planes. Como la proposición anterior puede ser (y de hecho es) muy dudosa y refutable - aunque tenga sus ventajas prácticas -, lo ideal es siempre visualizar al Sistema y a la

Planeación en forma Dinámica. Esto es, mediante Planes. A partir de aquí, en el desarrollo de esta tesis, siempre se utilizarán la palabra Plan o Planes.

1.1.3. Así, se recalca - en este contexto - que básicamente *Planear es escoger un Plan de acuerdo a cierto conjunto de especificaciones o criterios. Es decir, la actividad de Planear es análoga a la de resolver un problema* [1].

Ahora bien, el enfoque de *Planeación* depende de las especificaciones o criterios que se escojan, y , de entre los enfoques más utilizados los tres más relevantes son los siguientes :

La Planeación Satisfaciente

La Planeación Óptima y

La Planeación Adaptativa

Pero, antes de detallarlos, estudiaremos la Naturaleza de la *Planeación*.

[1] más formalmente, Planear es seleccionar un estrategia $E_i = (a_{i_1}, a_{i_2}, a_{i_3}, \dots , a_{i_k})$ en una malla de problemas. Donde la selección se efectú por medio de una evaluación de los E_i de acuerdo al conjunto de criteri elegido.

1.2 Naturaleza de la *Planeación*.

Como se ha mencionado, la *Planeación* es un proceso de Toma de Decisiones ; pero, no siempre una Toma de Decisiones es una *Planeación*. Su pecularidad - tal como lo define Ackoff (9) - se establece en tres sentidos :

1. Es una Toma de Decisiones anticipada.

2. Es un Sistema de Decisiones.

3. Es un proceso que se dirige hacia la producción de uno o más estados futuros deseables y que no es probable que ocurran a menos que se haga algo al respecto.

Puntos que se complementan con la definición de problema de *Planeación* que se exhibe en 1.1.1.

Antes de continuar con este análisis, se mencionarán algunos posibles tipos de *Planeación* , para así detectar más elementos aclaratorios.

1.2.1. Tipos de *Planeación* según su horizonte

Es común que se manejen las etiquetas de *Planeación Táctica* y *Planeación Estratégica*, pero estos conceptos rara vez se definen. Ackoff (9)

-- nos provee de una definición más operativa y clara :

> " *Las distinciones entre Planeación Estratégica y Planeación Táctica son más relativas que absolutas, esto es depende de la situación en particular puesto que mientras más largo e irreversible seal el efecto de un Plan, -- más estratégico será ; mientras más funciones de las actividades de una Organización sean afectadas por un Plan, más estratégica será. Si la Planeación se orienta más a los " fines " que a los " medios " más estratégica será. En cualquier otro caso será más Táctica* ".

Aunque dicha proposición de definición es más clara que la mayoría de las expuestas, contiene, también, muchos elementos de ambigüedad, que deben especificarse de una forma operativa y conveniente. Además no se debe - *Planeación Estratégica y la Planeación Tactica* - considerar separadamente Son complementarias. Se necesita tanto la *Planeación Estratégica* como la - *Táctica* para visualizar más adecuadamente - con máximo beneficio - todo el proceso, puesto que, considerarlas separadamente da lugar a los siguientes problemas y características :

	Planeación Táctica	*Planeación Estratégica*
EFECTOS DE UN PLAN	Más cortos y reversibles	Más largos, duraderos e irreversibles
INFLUENCIAS EN LAS COMPONENTES DE LA ORGANIZACION	Menores y de perspectivas más limitadas.	Mayores y de perspectivas más amplias.
HORIZONTE DE PLANEACION	Más a " corto " plazo, i.e. más eficiente en un horizonte de Planeación más limitado.	Más a " largo " plazo.
OBJETIVOS	Considera medios, fines, principalmente.	Considera metas y fines.
CASOS PROBLEMATICOS SEGUN UN ENFOQUE SEPARADO	Planeación apropiada para el corto plazo, no así para el largo plazo, teleológica a corto plazo	Planeación apropiada para el largo plazo, pero no para el corto plazo, teleológica a largo plazo.
CATEGORIZACION.	Corresponde a un segundo nivel de la red i.e., determina las actividades para implementar una estrategia.	Corresponde a un primer nivel de la red. Estudia y especifica las características de las estrategias para alcanzar el conjunto seleccionado de metas.

CARACTERISTICAS DE LA PLANEACION TACTICA Y DE LA PLANEACION ESTRATEGICA

CUADRO 1

CRITERIOS DE COMPROBACIÓN.

PLANEACION ESTRATEGICA	PLANEACION TACTICA
HORIZONTE DE PLANEACION :	HORIZONTE DE PLANEACION :
Máximo periodo relevante	Mínimo periodo relevante.
INCERTIDUMBRE :	INCERTIDUMBRE :
Mayor incertidumbre en el horizonte de Planeación.	Menor incertidumbre en el en el horizonte de Planeación.
PERIODO DE EFICIENCIA :	PERIODO DE EFICIENCIA :
Se configura la eficiencia de las alternativas implementadas en un horizonte de Planeación.	No necesariamente se configura , la eficiencia de las alternativas implementadas más allá del horizonte de Planeación. Puede ser mas fácilmente corregible dentro del horizonte de Planeación. En el caso en que la solución sea óptima en el horizonte de Planeación, pero que más allá sea incluso no factible, la posibilidad de solución disminuye.
Si la relación de los elementos del ambiente es conflictiva u hostil la posibilidad de implementación del plan disminuye. Sus costos directos son mayores.	Si no se revisa continuamente - puede producir soluciones " parches " , que irían en contra del enfoque Sistémico. Tiende a producir un costo de - oportunidad global más importante. Si hubo error de identificación en los clientes, muy probablemente la organización está en una - situación " más confusa " que antes de haberla aplicado. Si hubo falla en la detección del periodo relevante y éste es menor de lo que convendría, es posible que se sobrestime el valor de la alternativa de solución elegida.

CUADRO 2

En suma, en mi opinión, para evitar los errores de considerarlos separadamente, se debe especificar que no son dos tipos de *Planeación* distintas, - sino de dos componentes del marco de *Planeación*.

En esa forma, podemos concluir que toda *Planeación* debe ser *Estratégica y Táctica* a la vez, y que se deben delimitar adecuadamente un conjunto de aspectos de *Planeación*, como los siguientes :

a) *Horizonte de Planeación.*

b) *Unidades de Planeación.*

c) *Fines.* Especificando metas, objetivos e ideales (pág. 56)

d) *Medios.* Elección de políticas, subplanes, estrategias, programas, procedimientos y prácticas con las que habrán de alcanzarse los objetivos.

e) *Recursos.* Determinación de tipos, cantidades, características de los recursos que se necesitan, defi - niendo como se habrán de asignarse a las actividades.

f) *Realización.* Diseñar los procedimientos para la Toma de Decisiones, considerando la coordinación inte - gral sobre la implementación.

g) *Control.* Procedimiento para prever o detectar los errores o las faltas de la *Planeación*, así como - para prevenirlos o corregirlos sobre una base de continuidad.

De esa forma, para seleccionar un plan de acuerdo a los elementos de trabajo anteriores, se necesita estudiar las escuelas de *Planeación*, que son combinación de enfoques que pueden representarse - según Ackoff - como la *Planeación Satisfaciente, Planeación Óptima y Planeación Adaptativa* - (9).

1.3 LOS ENFOQUES DE PLANEACION

1.3.1. LA PLANEACION SATISFACIENTE

> " UN PLAN FACTIBLE, AUNQUE NO SEA EL MEJOR, ES MEJOR QUE UN PLAN ÓPTIMO PERO IRREALIZABLE "

Satisfacer, es hacer algo " *bastante bien* ", pero no necesariamente " *mejor que se pueda* ". El nivel de realización - o la medida de " *satisfacción* " - la establece quien toma la decisión (T. de D.). (ver " U Concepto de Planeación de Empresas " Ackoff, R.L.)

En este tipo de *Planeación* se declaran los objetivos y metas (formuladas) en términos de medidas de rendimiento. No se busca establecerlas " *tan altas* " como sea posible, sino simplemente " *suficientomente altas* " y sólo se revisarán si resultan inaccesibles (en la implementación). Al fijar los objetivos y metas, se busca sólo un medio aceptable y factible para alcanzarlos (no necesariamente el mejor medio posible

El Planificador Satisfaciente parece operar sobre el proncipio de que si uno puede medir lo que se quiere, debería, o bien desea únicamente lo que puede medir, o no tratar de medir.

Trata de " *maximizar* " su factibilidad - pocas veces definida explícitamente - por medio de :

a) Minimizar el número y la magnitud de desviaciones de las prácticas y políticas en vigor,

b) Especificar, a los sumo, modestos incrementos en la necesidad de recursos, y

c) Efectuar cambio mínimos en la estructura de la Organización.

Debido a lo anterior (especialmente por (b)) se presupone la consideración de los componentes de *Planeación* como relacionados " moderadamente " , aunque algunas veces no especifiquen las relaciones de una manera explícita. Por ejemplo, el caso de un paciente que tiene varias dolencias y el médico procura ajustar se estado de salud de tal forma que uno de sus padecimientos (digamos, el más grave) quede cubierto dentro de lo más posible, y le recomienda seguir al pie de la letra las indicaciones previas (dietas, acondicionamiento físico, reposo, etc.).

En cierta forma la mantiene en un estado que, si bien no es de salud ópt ma, si es de " mejoramiento ".

El problema que se desprende es que si no existe control y una constante evaluación del estado de salud del enfermo, el mejoramiento es sólo temporal y la <u>recaída puede ser peor</u>. (por muchas razones. entre ellas qu el tiempo perdido hace que la posible recuperación sea más tardada).

Por otra parte, si se desea efectuar tal sólo cambios mínimos en la Organización (i.e. en este caso, en el régimen de vida y conducta), se presupone una <u>garantía de confiabilidad</u> en la Organización previa que - sería discutible. Ahora bien, si esos aspectos se consideran mejorables - elección que depende del criterio del Planeador - se llegaría a un -- reacondicionamiento mejor (más integral), y muy posiblemente más duradero.

Además, presupone que la garantía de confiabilidad (y la de <u>estabilidad</u>) de la Organización en lo futuro se mantiene. La *Planeación Satisfaciente* no produce cambios radicales a lo ya existente. Tal como menciona Ackoff (9) " *normalmente es el reflejo de planes conservadores que continúan cómodamente con la mayor parte de las políticas en uso, corrigiendo solamente deficiencias notorias* ". Cuando el estudio y evalua ción de esas deficiencias no se establecen en forma Sistemática, se corre el riesgo de que se <u>resuelva</u> solamente en situaciones críticas para la - <u>supervivencia</u> de la organización (y no se tiene, entonces, como objeti-

-- vo el preguntarse por el desarrollo y crecimiento, tal como lo hacen otros tipos de *Planeación*); en ese sentido, la *Planeación Satisfaciente* considera implícitamente :

1.3.1.1. (1) Un nivel de estabilidad de la Organización y del ambiente en donde está embebida,

(2) Un enfoque optimista en donde se supone que al contar con un recurso principal (comúnmente suficientes recursos monetarios) cualquier otro tipo de recurso podrá adquirirsecuando se necesite,

(3) Que es preferible no modificar la estructura de la Organización, pues toda solución al problema de *Planeación* debe ser " sencilla "

Esos puntos despiertan las criticas básicas de la *Planeación Satisfaciente*:

1.3.1.2. (i) Si el Sistema es altamente dinámico o su comportamiento está afectado por una sucesión de cambios relevantes muy rápidos y de cierto nivel de intensidad, la *Planeación Satisfaciente*, desafortunadamente muy usada sin descriminación en nuestro país, ocasiona las soluciones " parches " que, en el mediano o largo plazo. puede ser extremadamente ineficientes.

(ii) Otro problema consiste en la falta de discriminación y evaluación de objetivos cuya consistencia sólo se descubre al implementar la solución, donde, nuevamente, puede ser tardía.

Desde luego, dichas actividades de *Planeación* implican un costo mayor y el no realizarlas directamente causa que los costos directos sean menores y la *Planeación Satisfaciente* sea más barata (con el criterio de insumos principalmente), ésta es una de las razones de que sea preferida. Aún más, si la responsabilidad en el Proceso de *Planeación* se diluye, entre otras razones por efectos políticos del Sistema, o porque establecen cambios en prácticas realizadas con anterioridad, es peligroso para la supervivencia política del Planeador; la elección de la *Planeación Satisfaciente* es más que obvia.

(iii) El problema del Control : Dada la suposición de que si surge lo inesperado, la Organización será capaz de resolverlo, al usar este enfoque pocas veces se diseñan Sistemas formales para controlar un Plan después de haberse implantado, con lo que se confirmará, en los casos en donde no existe control, que no existe *Planeación*, sino simplemente algunos aspectos de ésta que, desde luego, aumentan la inversión y ocasionan erogaciones (altas, aunque sean pequeñas en el corto plazo) que pueden no ser siquiera paliativos de los problemas en el Sistema Social.

1.3.2. LA PLANEACION OPTIMA.

" SE HACE UN ESFUERZO POR HACER LAS COSAS NO SOLO SUFICIENTEMENTE BIEN, SINO POR HACERLAS LO MEJOR POSIBLE "

El Planificador Optimizador trata de formular metas en términos cuantitativos y de combinarlos en una medida única de rendimiento para toda la -- Organización. Algunas veces puede transformar una variedad de metas en - medidas en una escala única (generalmente monetaria), y, por lo tanto puede combinarlas en una medida general de eficiencia; por ello, generalmente se cae en el error de pasar por alto las metas que no puede cuantificar [2].

Casi siempre, los intentos por optimizar la estructura requieren un uso - extensivo de juicios cualitativos y de su transformación en medidas cuantitativas, ocasionando interferencias, puesto que, la matematización puede ser muy subjetiva y por lo tanto restrictiva.

Los Optimizadores elaboran Sistemas de control que puedan detectar y corregir los errores vistos con anticipación (con respecto a la representación de la realidad escogida).

[2] Así produciendo un Modelo que es, entonces, una representación limitada y parcial de la realidad social con todas las implicaciones que eso conlleva.

Dichos Planificadores buscan los mejores planes, programas, procedimientos y prácticas posibles por medio del uso de Modelos Matemáticos. Sus éxitos dependen de que el Modelo sea una *representación lo más completa y fiel del Sistema, y las soluciones del Modelo sean factibles, una vez que éste se ha elaborado.*

Los esfuerzos por desarrollar Planes Optimos casi siempre producen un resultado valioso, (aun los esfuerzos sin éxito) y una comprensión más completa del Sistema para el cual se Planea.

El Optimizador trata de :

a) Minimizar los recursos que se necesitan para obtener un nivel especifico de rendimiento,

b) Maximizar el rendimiento que se puede conseguir con recursos disponibles (o que se espera que lo sean) o,

c) Tener el mejor equilibrio entre los costos (recursos consumido) y los beneficios (resultados). Por ejemplo, el enfoque de Costo- Beneficio.

Los Planificadores Optimizadores, no suelen tratar explícitamente la Estructura de la Organización [3], porque apenas se ha comenzado a desarrollar los Modelos que se refieren a ella. (El objetivo de la *Planeación* de la Organización es crear una Organización que pueda lograr de manera efectiva objetivos generales y, especificamente, realizar los Planes que produzcan).

Sin embargo, la realización de un Plan nunca debe ser mecanicista, puesto que, depende de la buena voluntad y cooperación de los individuos y grupos en la Organización, y entonces se presenta una falla del enfoque de *Planeación Óptima*, sobre todo porque una mala identificación de la Organización llevaría a un Modelo cuya solución óptima no es " realmente " solución.

Las técnicas de Optimización han sido más útiles en la *Planeación Táctica* que en la *Planeación Estratégica* (por las dificultades de evaluar futuros lejanos y los efectos a largo plazo), aunque puede ser sumamente --- útil la *Planeación* que optimiza las partes de un Plan y las integra con las demás partes que se han tratado con otras técnicas que no son de Optimización ; dicho de otra manera, un procedimiento de *Planeación Cualitativa Satisfaciente* - *Integral o Sistémico* - probablemente produzca mejores resultados que los que pueden alcanzar la *Satisfaciente* o la *Optimizadora* en forma aislada.

[3] Estructura de la Organización; se entenderá como la forma en que se divide el trabajo que desempeña una Organización (es decir, por funciones, como Producción, Región Geográfica, Mercadotecnia) y cómo se asigna el trabajo a las componentes de la Organización.

En cuanto a la relación del Planificador con la Estructura Organizacional se insistirá que el Planificador Optimizador, generalmente, da por supuesta la estructura del Sistema y busca una estrategia que mejor resuleva el problema.

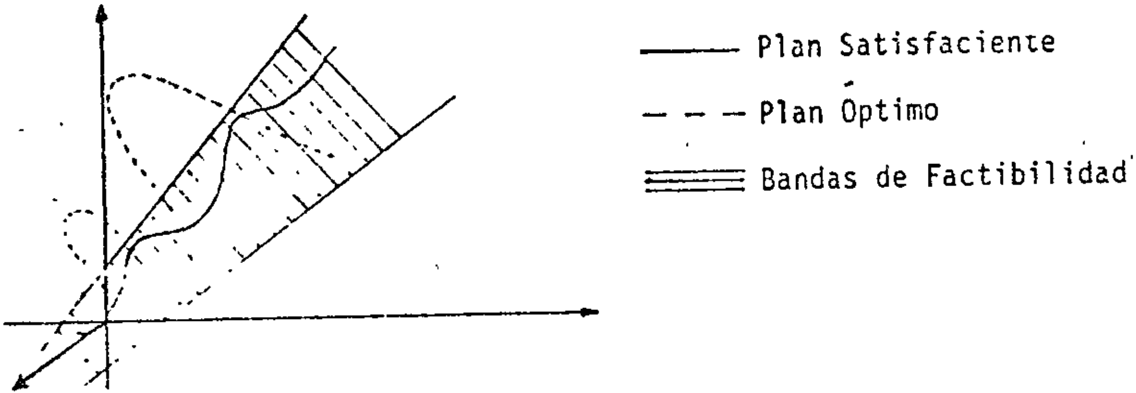
Para poder captar de manera más clara los puntos de la *Planeación Óptima* y compararla con la *Planeación Satisfaciente*, se propondrá el siguiente ejemplo :

El mismo paciente que antes mencionamos acude a un Centro de Detección -= Médica - v.gr. a un Centro Médico Computarizado - a través de los instrumentos de diagnóstico se le completa su cuadro clínico (que es de - todas forma parcial, pues algunas funciones relevantes no son todavía - mensurables) y se identifica su enfermedad según clasificación previa, con cierta tipología. De acuerdo con la analogía y experiencia, se le da tratamiento, pero, si las funciones relevantes no detectadas no fueron - consideradas (por ejemplo, aspectos psicológicos) la eficiencia del Modelo - y del tratamiento- no es tan efectiva como se considera. Aunque en esta ilustración. el Modelo de Optimización no corresponde estricta - mente a un Modelo Matemático, coincide con los supuestos subyacentes del Enfoque de *Planeación Óptima* y sirve para representar la " debilidad " de la cuantificación en aspectos no claramente cuantificables.

Adémas, exhibe el hecho que se mencionaba, ya sea, que podría ocurrir que la *Planeación Satisfaciente* (o un Plan Satisfaciente) sea mejor que la *Planeación Óptima* (o que un Plan Óptimo)[4], tal como se observa en los siguientes gráficas :

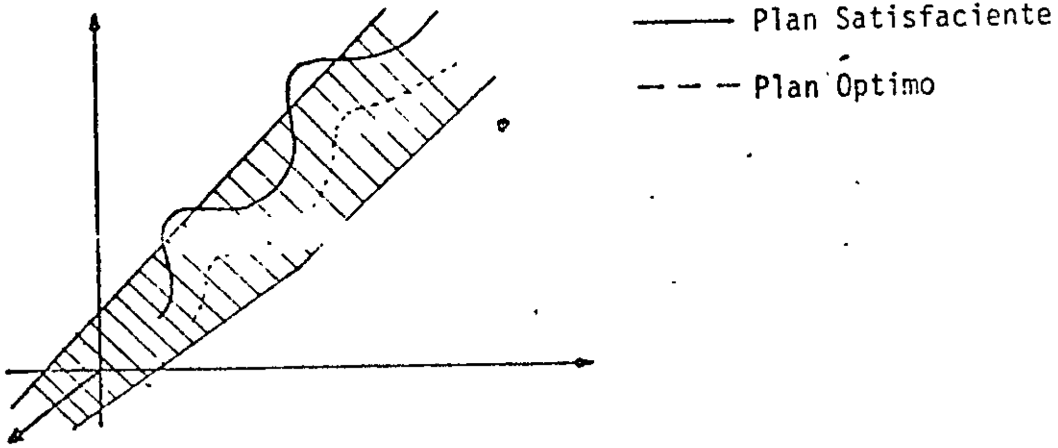
GRAFICA 1

EFICIENCIA SEGÚN EL MODELO



GRAFICA 2

EFICIENCIA REAL



[4] Todo porque la información del Plan Satisfaciente sobre la Organización es más relevante y adecuada que la utilizada por el Plan Óptimo.

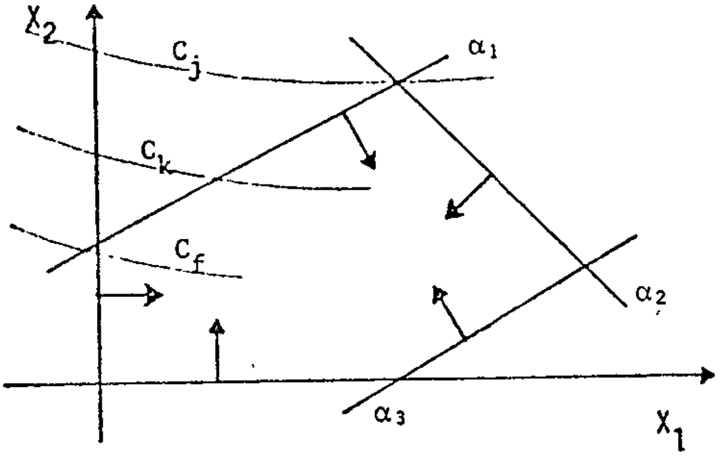
Por otro lado, se preguntará el que si el Planificador Optimizador generalmente da por supuesta la Estructura del Sistema y solamente busca una estrategia que mejor resuelva el problema, existirá la opción a que se cuestione sobre la validez de la Estructura y de las funciones correspondientes, por ejemplo, en el caso de la Planeación Nacional ¿ Existirá la opción de que se pregunte si el gobierno es capaz de crear suficientes fuentes de trabajo, que produzcan riqueza y sirvan de vehículo para distribuirla ? o tan sólo ¿ formulará un Modelo que estudie el beneficio social o la utilidad y así escogerá una estrategia que lo maximice ?.

¿ i.e. inquirirá sobre premisas básicas de Estructura y finalidad o solamente optimizará el funcionamiento de una estructura dada ?

GRAFICA 3

ESQUEMA DE UN PROBLEMA DE PROGRAMACIÓN MATEMÁTICA

(caso de dos variables de Decisión)



donde,

Las C_{k_j} denotan contornos de la función objetivo.

Las α_{i-s} indican el Hiperplano soporte de cada una de las restricciones.

a) C_j indicaría la solución óptima (Criterio de Optimalidad Matemá - tica) [5]

b) C_k , C_f indican algunos factibles.

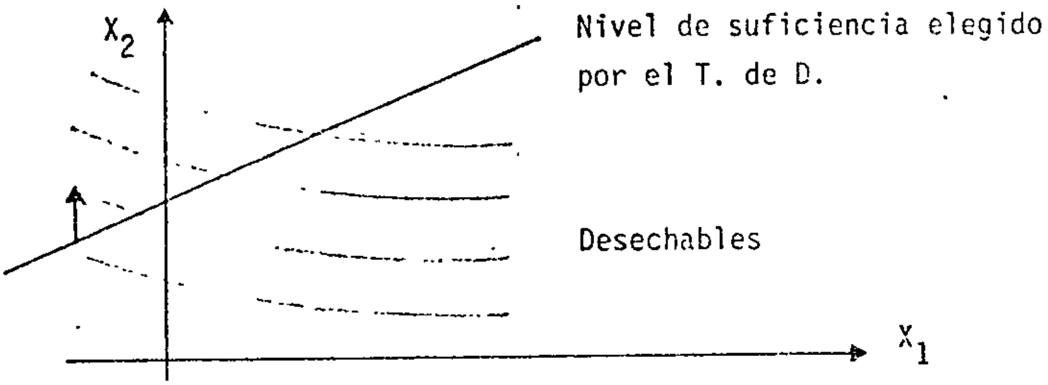
[5] En el contexto de que un punto óptimo es necesariamente factible, i.e. $O \subset F$ (Optimas ⊂ Factibles)

COMPARACIÓN DE CRITERIOS

(Por Diagramas)

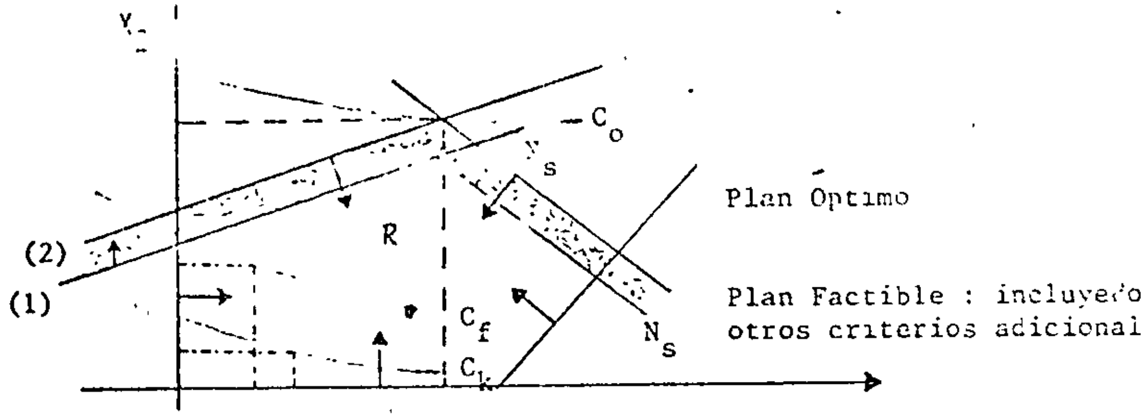
GRAFICA 4

CASO I MÍNIMO NIVEL SATISFACTORIO



GRAFICA 5

CASO II [illegible] Y PLANES FACTIB[illegible] [illegible] TÉRMINOS DE LA APLICACIÓN



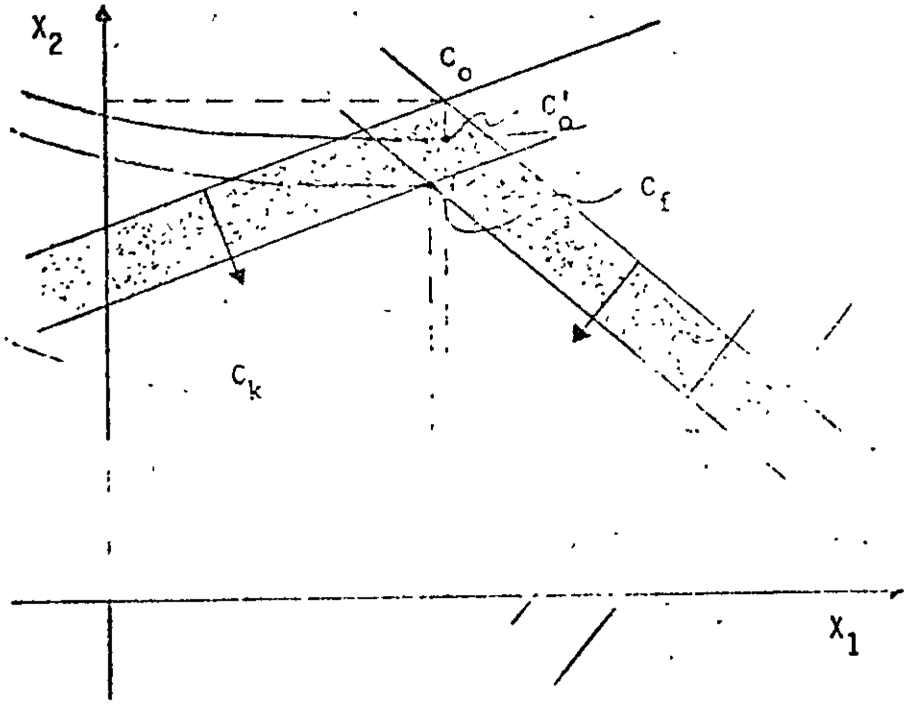
(1) En el mínimo nivel satisfactorio según el esquema pesimista se determina un nivel máximo donde se puede llegar fijado por criterios adicionales (políticos, sociales, ideológicos, etc),

(2) Máximo nivel de Satisfacción factible.

Axioma Ackoff

" Es mejor la solución aproximada de un Plan Óptimo realizable que la solución exacta de un Plan factible ".

- GRAFICA 6 -



1.3.3. LA PLANEACIÓN ADAPTATIVA.

" EL PROGRESO ES NUESTRO PRODUCTO MÁS IMPORTANTE "

La Planeación Adaptativa o Innovadora. Los Planes de este tipo se pueden diseñar en las Organizaciones en un grado más amplio de lo que se pudiera pensar por las prácticas actuales.

Tiene tres bases :

1. Se basa en la creencia de que el valor principal de la *Planeación* no descansa en los planes que se producen, sino en el proceso de producirlos.

   (La *Planeación* efectiva no puece hacerse para una organización, sino que debe hacerse por la gente que la forma).

2. La necesidad real de la *Planeación* obedece a la falta de administración y cOntroles efectivos. El principal objetivo de la *Planeación* debería ser la proyección de una Organización y un Sistema de Administración que minimice la necesidad futura de la *Planeación Retrospectiva* - esto es, la *Planeación* encaminada a corregir las deficiencias producidas por decisiones hechas anteriormente - y lograrlo reduciendo las posibilidades de que ocurran tales deficiencias. El objetivo

es considerar relevantemente la *Planeación Prospectiva* : *La Planeación* que se dirige a crear un futuro deseado.

3. El conocimiento que tenemos del futuro se puede clasificar en : *Certeza, Riego e Incertidumbre* (1) cada uno de ellos requiere un tipo distinto de *Planeación de Compromiso o Certeza, Contingencia y Sensibilidad.*

a. *Certeza.* Existen ciertos aspectos del futuro sobre los que se puede tener una virtual certeza, y podemos establecer una *Planeación de Compromiso*. Aun aquí la posibilidad de error [illegible] en consideración al establecer controles apropiados. Para ello, se requiere la actualización continua de las estimaciones de los que es inevitable o invariable. La prudencia nos indica que para alcanzar el objetivo deseado, no se deben contraer compromisos antes de lo indispensable.

Así, la *Planeación* que potencialmente produce mejores resultados a largo plazo, considera :

1. Descubrir lo inevitable ;
2. Determinar como explotarlo y enfrentarlo ; y
3. Acreditarlo a quien así lo haya hecho.

b. *Riesgo*. Existen ciertos aspectos del futuro sobre los -- cuales no podemos estar relativamente seguros, - pero podemos asegurarnos razonablemente de cuáles son sus probabilidades. Se necesita una *Planeación Contingente* ; es decir, deberíamos preparar un Plan por cada probabilidad, para así poder aprovechar rápidamente las oportunidades que se presenten cuando " *se decida el futuro* " [6].

c. *Incertidumbre*. Existen ciertos aspectos del futuro que no podemos prever ; por ejemplo, las catástrofes naturales o políticas o los avances tecnológicos. Podemos prepararnos para ello de manera indirecta por medio de una *Planeación Reactiva*, que está encaminada [illegible] diseño de una Organización y de un Sistema [illegible] de detectar desviaciones de la ruta asignada y poder reaccionar ante ellas de manera efectiva.

La adaptación es una respuesta a un cambio (estímulo), que reduce real- y potencialmente la eficiencia de la conducta de un Sistema; una respuesta que evita que ocurra tal reducción. El cambio, puede ser interno (centro del Sistema) o externo (en su ambiente).

[6] y donde se pueden utilizar los llamados Escenarios (ver pág.39)

Existen dos tipos de reacciones adaptativas : La Adaptación *Pasiva*, en la cual el Sistema cambia su conducta mejorando su rendimiento en su medio cambiante, y la Adaptación *Activa*, en la cual el Sistema modifica su ambiente, de manera que, su propia conducta (presente o futura) sea más eficiente.

El Planificador Adaptador, trata de cambiar el Sistema de manera que la conducta más eficaz aparezca " *naturalmente* " .

Dicho tipo de *Planeación* nos permite un medio efectivo de manejar las variaciones del Sistema a largo plazo como a corto plazo.

1.3.4. Siempre se buscará el diseñar una *Planeación Integral*, *Planeación Eficiente*, o *Planeación Direccionada*.

Las etiquetas pueden ser muy variadas, y pueden existir diversas diferencias entre un concepto de *Planeación* y otro. Por ejemplo, la *Planeación Integral*, que tendría como componentes a la *Planeación Estratégica y la Planeación Táctica*, tiene, en teoría, una satisfacción *Teleológica* tanto a largo como a corto plazo; se le coloca mecanismos de control y realización; se da un proceso de evaluación; se utilizan técnicas de previsión; y se espera formular un mecanismo de coordinación integrada.

Si bien es cierto que la *Planeación Adaptativa* está aún en sus inicios, --- presenta las siguientes ventajas :

(1) Define y aclara el concepto de Adaptación;

(2) Distingue entre la adaptación pasiva - propia de Sistemas que son flexibles - que es inherente al Sistema en términos de la modificación de su conducta y funciones, mejorándolas como respuesta a estímulos cambiantes provenientes del Ambiente [7] ; por ejemplo, en el caso de una empresa que modifica los precios o el tipo de sus productos debido a la competencia en un Sistema capitalista o a efectos de la demanda del producto (aunque esta situación -- puede presentar una combinación de los tipos de adaptación caracterizados), como también en el caso del paciente que decide cambiar dietas alimenticias por influencia del médico o de la familia o su manera de pensar hacia los médicos por efecto del trato favorable de alguno de ellos. Por --- otra parte,en la adaptación activa el Sistema cambia su -- ambiente de manera que su propia conducta presente o futura sea más eficiente. Por ejemplo, el caso del paciente -- que decide cambiar su ocupación para ajustarse a las necesidades de su enfermedad; o el de la empresa que se dirige a otro tipo de clientes cambiando su mercado. En otro ----

[7] relacionada con el efecto del aprendizaje.

contexto, el campesino que decide incorporarse a una Organización colectiva en la producción agropecuaria ante la presión del capital o los grandes terratenientes y así evitar la depauperización, está intentando cambiar su ambiente.

Desde luego, las adaptaciones de la Organanización se presentan muchas veces mezcladas y, deslindarlas puede ser, en algunas de ellas, algo muy complicado, (puesto que, en algunas ocasiones, aunque la intención original sea modificar la conducta del Sistema, el ambiente se modifica).

En la *Planeación* se necesita de mecanismos de identificación para hallar la respuesta adecuada, y, de control, para regular la intensidad de la misma. Tal característica requiere de flexibilidad, quizá la propiedad más importante en las Organizaciones y descarta la rigidez como propia de Sistemas en los que estímulos de bajo nivel e intensidad no son aceptados (i.e. -- " rebotan ") en el Sistema y son casi nulas las reacciones de los componentes hacia el Ambiente. Así lo confirman Emery (31) y, en el caso de los Sistemas políticos tanto Easton (30), como Young (65). Algunas ocasiones, la *Plàreación Adaptativa* se ha descrito como un " *un proceso evolutivo que empieza con especificaciones parciales sucesivas que se -- van adoptando mediante articulación progresiva, revisión y ajuste a las situaciones cambiantes del ambiente* " (55). De esta forma, las adaptaciones, tanto pasivas como activas, se regulan dentro del mismo [illegible] por medio de " *agujas* " (dispositivos) de detección. Un ejemplo, se espera, aclara la situacion :

Supóngase el mismo caso del paciente anteriormente mencionado, en el que el diagnóstico médico ha detectado problemas en el hígado, corazón, riñones y problemas en la digestión, entonces, opta por adecuar al tratamiento todos los órganos (las funciones) en forma Integral, pero rehabilitándolos ta rápido como se pueda, buscando lograr que, mediante el tratamiento, los órganos digestivos vayan aceptando sustancias (cada vez más poderosas) necesarias para el funcionamiento del corazón (i.e. está presuponiendo que existe una interacción entre todos los componentes). Así, avanza progresivamente : realiza exámenes médicos (evaluaciones) constantes para observar el efecto de su medicación y, de esa forma, poder detener y cambiar, si es el caso, la dirección de mejora en algún órgano. Bien puede ser, desde luego, que existan fallas importantes en el funcionamiento de algún órgano, por ejemplo el hígado, pero estará en mejor posición para resolverlos. Dado que no existe aún, desafortunadamente, el medicamento universal y la extirpación, no es " la solución " este procedimiento - ejemplo de la *Planeación Adaptativa* - se dirige a :

(a) ganar tiempo en la detención de la red de causas e interrelaciones de la enfermedad,

(b) a mejorar (rápidamente) por niveles las funciones (órganos) del paciente <u>sin</u> descuidar alguna. Es obvio qu la mejoría en varias direcciones no tendrá la misma velocidad, pero el tratamiento (y éste puede ser modificabl i.e. flexible) podrá llegar al punto en que sea el <u>organismo completo</u> el que avance hasta su recuperación.

A cada componente de la Organización corresponderá una aguja detectora que formara la *red de Planeación*. Dicha red necesita especificarse en forma -- Integral.

Definir la *Planeación Integral* en términos generales tropieza con el hecho de que - por la existencia de varias escuelas y muchas versiones - existen numerosas definiciones sin nada en común; más aún, se ha definido la - *Planeación Integral* de la Educación (52), de los Transportes, etc. que según la *Teoría de Sistemas* serían todo, menos integrales, v.gr. en el caso de la *planeación* educativa se toman en cuenta - a grandes rasgos - (1) la futura demanda de vacantes, considerando las tendencias demográficas y sociales; (2) se establece la relación de la educación con la economía; (3) se evalúan los sectores productivos que bajo las políticas de desarrollo rindan más. Aun en los países socialistas donde se busca, al menos teóricamente, distribuir más prudente y equitativamente los recursos humanos y materiales, la *Planeación* " *Integral* ". de la educación no es todavía integral: los efectos sociales, el desarrollo demográfico y los asentamientos humanos, la movilidad social y la estructura política, etc. no son nada más causas o efectos de la educación, sino que existen interacciones dobles y, desde luego, indirectas. La acepción seleccionada de *Planeación Integral* es equivalente a '*Sistémica* - i.e. usando la *Teoría de Sistemas* - deben considerar todos los componentes del Sistema y su relación con el Suprasistema. *La Planeación Integral*, asimismo, es interdisciplinaria; en forma profunda, actúa en el marco del Método Científico y es, por lo tanto, el área de acción de la Investigación de Operaciones (ver 24).

En cuanto a la colección de problemas de *planeación* (i.e. en la parte -- prospectiva) se tiene, según se menciona anteriormente, tres tipos de la misma :

(1) de Certeza - que Ackoff denomina de <u>Compromiso</u> (1)

(2) Contingente

(3) Reactiva

Esa clasificación es una analogía con la que se establece en cuanto a las situaciones problemáticas según la relación entre alternativas $\{ a_i \}_i$ y resultados $\{ o_j \}_j$.

(Problemas de (1) certeza, (2) riesgo e (3) incertidumbre).

(1) Cuando se modelan ciertos aspectos del futuro como casi - ciertos (con certeza virtual), se tiene una *planeación de compromiso*, por ejemplo, cuando en una población estable se buscan estimaciones sobre el número de personas de ciertas edades, se pueden formular con gran precisión. Naturalmente que, aunque en términos de los Modelos de *planeación* (especialmente utilizando la *planeación óptima*) la facilidad de manipulación sea mayor, las limitaciones - que surjan de analizar situaciones dinámicas (e inciertas) como casi seguras, crean compromisos con las previsiones que, se divergen significativamente con la reali

-- dad, ocasionan que los planes propuestos sean ineficientes.

(2) Cuando el futuro presenta eventos de los cuales no podemos evaluar las probabilidades de ocurrencia de los posibles - resultados, se emplea una *Planeación Contingente*, por lo - que, se prepara un plan para cada posibilidad y así permite aprovechar rápidamente las oportunidades en lo futuro. Esta *planeación* trata de abarcar todas las posibilidades - <u>anticipadamente</u> para decidir tan pronto una posibilidad -- (o resultado) se convierte en realidad. En esta tesis - se hace hincapié en la gran [illegible] de la [illegible] *tingente* y de los escenarios como [illegible] previsión y, consecuentemente, de la *planeación*.

[8] Escenarios. " The imaginative construction into the future of a logical sequence of events based on specified assumptions and -- initial conditions in a given problem area ".

Un enfoque adecuado para adivinar el desenlace de una situación dada sería lograr el diseño de Escenarios, o sea posibles capítulos de " historia futura " ; un escenario no es en sí mismo un pronóstico de los que acontecera en la realidad, sino, más bien, un elemento de apoyo para visualizar los problemas con mayor nitidez, a fin de poder adaptar las decisiones más acertadas.

Los escenarios frecuentemente se antojan inverosímiles por pintar acontecimientos específicos, cuya probabilidad de realización es infinita - mente pequeña, pero la historia está hecha de una cadena de acontecimientos que pocos años hubieran parecido poco probables.

(3) Finalmente, existen aspectos del futuro por completo imprevisibles (casi todos en el largo plazo), i.e. inciertos, a los que se enfrenta el Sistema de una forma indirecta; - ése es el caso de la llamada *Planeación Reactiva*, que se - dirige hacia el *diseño de una Organización y de los instrumentos de control y regularización*. En este caso, la retroalimentación es indispensable y la red de agujas de detec -- ción así lo hace ver. La diferencia más notable entre las - tres reside en que, obviamente, toda previsión puede hacerse de manera reactiva, ganando en información relevante --- (mediante costos adicionales) se puede transformar en -- *Planeación Contingente* (como cuando se pasa de un problema de Incertidumbre en uno de Riesgo) y que, en casos muy raros, cuando el futuro relevante es casi completamente previsible, se usa la *Planeación de Certeza*; de manera natural, existiría una posibilidad mayor para la *planeación de certeza* cuando el horizonte de *planeación* es a corto plazo. - Sin embargo, aun en este caso hay que medir la posibilidad de error en el Modelo de *Planeación*, lo cual puede ser bastante difícil.

En particular, en la *planeación social* se sugiere considerar exclusivamente la *planeación contingente o la planeación reactiva*, y así se ilustrará en el caso de las comunidades en estudio.

Finalmente, existen dos conceptos adicionales que son necesarioa para la discusión posterior :

1.3.5. La llamada *Planeación Dialéctica*, que consiste en la generación de los " mejores " planes contradictorios que representan valores y puntos de vista conflictivos, los que se presentan a un debate estructurado, usando los mismos datos iniciales para cada visión en discusión o debate exhaustivo. (Por ejemplo, el enfoque de Mitroff (47), en el caso de la *Planeación Dialéctica Estratégica*).

La Planeación Normativa, que también se denomina *Planeación Teleológica*, en la que se examina críticamente los juicios de valor fundamentales subyacentes en las metas y objetivos de *planeación*.[9]

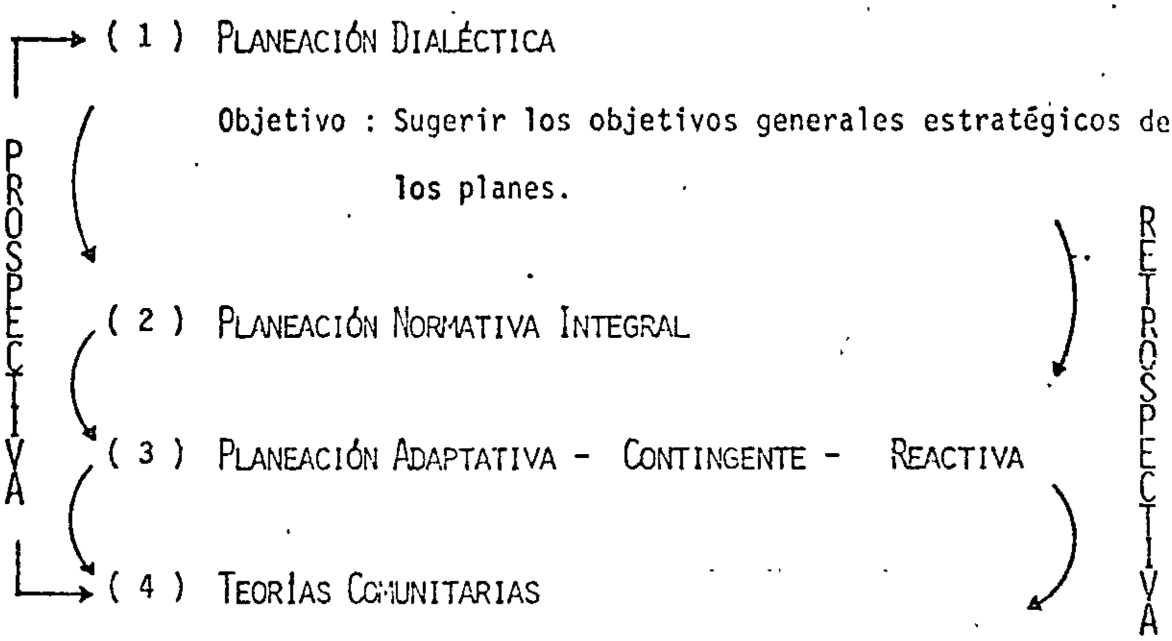
De ahí que se propone el siguiente esquema general de trabajo por niveles :

[9] La cual tiene como componentes :

(1) La Planeación Estratégica : con referencia a metas específicas; y

(2) La Planeación Táctica que se dirige a preparar la obtención de las metas previamente definidas.

- DIAGRAMA 2 -



(1) PLANEACIÓN DIALÉCTICA

Objetivo : Sugerir los objetivos generales estratégicos de los planes.

(2) PLANEACIÓN NORMATIVA INTEGRAL

(3) PLANEACIÓN ADAPTATIVA - CONTINGENTE - REACTIVA

(4) TEORÍAS COMUNITARIAS

PROSPECTIVA

RETROSPECTIVA

No obstante, no se puede perder de vista, como ya se mencionó anteriormente, que el enfoque de *Planeación* está referido a una sucesión de problemas.[10] Todo enfoque de *Planeación* debe declarar los principios ideológicos que lo sostiene, cuestionar si es legítimo - y por qué - ayudar a los beneficiarios propuestos, cuales serían sus responsabilidades e identificar si la -- aplicación de la solución llevará, nuevamente, a una manipulación interesada e impuesta de la Comunidad o a imponer soluciones a problemas que no reflejan la problemática real de la misma (y así cometer el error tipo III: " *resolver el problema equivocado* " (47). Ningún aspecto de los anteriores debe soslayarse y el papel del planeador, con referencia a su medio

[10] El tipo de problema, la Organización que se estudia, la clase de clientes o beneficiarios y el rol del Planificador son algunas de las variantes posibles.

social debe, necesariamente, ser analizado. En conclusión, como se examinará después, *seguir el Enfoque de Planeación es necesario, pero no suficiente -- para la actividad de Planeación.*

INSIDE:
Planning Models, Strategic Models, Operational Models, Function Relationships, Data Relationships, Model Maintenance, Case Study

Structured Systems Planning

PAYOFF IDEA. Because of the steadily rising cost of information systems, interest in systems planning is increasing. A tremendous amount of effort can be wasted in developing a systems plan, especially if the effort is haphazard or if the desired output documents are not predetermined. This portfolio discusses a systematic approach to systems planning and presents a case study of the development of a systems planning methodology.

PROBLEMS ADDRESSED

There is some similarity between the systems planning procedure discussed in this portfolio and that used to build the often disastrous mammoth MIS systems of the late sixties. Reactions to those failures included the abandonment of planning and the growth of interest in decision-support systems and in relational data bases. If the sixties were overly optimistic, the seventies leaned toward resignation.

However, the development and maintenance of information systems has become too expensive for these approaches to be completely effective. Many systems that take more than five years to develop are dated before they are implemented. Large, expensive data bases contain far more data than necessary, not because of poor design but because the link to corporate needs is unclear. The design practices of the sixties that compelled system designers to leave blank fields in records for future enhancements have evolved into the current practice of adding extra data that may be needed at some time in the future. This is easy to rationalize because the data base system can handle the additional data.

Tools that focus data processing efforts help resolve these problems. The planning process described in this portfolio helps bridge the gap between information systems expectations and results. This procedure can substantially reduce systems development time and reduce the credibility gap between the DP and user departments.

PLANNING MODELS

Most planning methodologies develop a model of an organization's information systems. Problems are encountered in implementing these models because:

- They are time-consuming to develop.
- It is difficult to translate the models into definitions for systems applications.
- They may not directly relate to business plans.
- It is hard to maintain the documentation; thus, the models fail to evolve.

Nonetheless, developing good models is vital to translating strategic plans into operating plans.

Two types of models, each equally important, must emerge from the planning process: a function or process model, which describes major functional responsibilities, and data models, which describe significant data hierarchies.

Several types and levels of models should be used in the successive refinement of strategic plans into executable application projects. Three levels are discussed in this portfolio, although in practice the number of levels depends on the needs of the organization. The three levels discussed are:

- Strategic models
- Application group models
- Application systems models

Strategic Models

At the strategic level, a general model is needed to provide a high-level framework. "The network should provide a visual depiction of the strategic long-range objective of the information systems plan The network is the logical relationship of information systems (and their external interfaces) deemed necessary to support the information requirements of the organization [3] " I [illegible] illustrates an information resource model developed for Oglethor, [illegible] ower Corporation. Much of this portfolio is based on experiences at Oglethorpe (see case study later).

Although this high-level network model is important, it needs to be supplemented, even at the strategic-planning level, with additional detail. These details are shown in:

- An entity or context diagram that defines the major external organizational interfaces (see Figure 2)

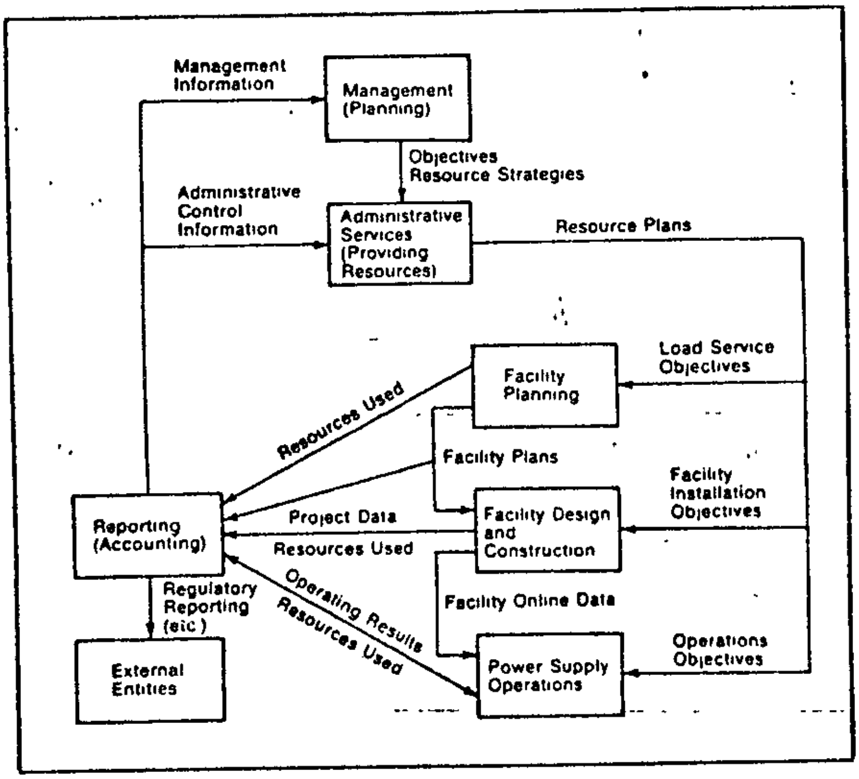


Figure 1. Oglethorpe Power Corporation Information Resource Model

- A more detailed diagram that expands each of the network's major functions into subfunctions and data bases (see Figure 3)

Preparing these diagrams enables the planner to focus on key business functions and types of information. Although very high level, they provide a framework for further analysis. Two points to concentrate on during this initial work are the functional rather than the organizational breakdowns (not always easy) and any anticipated new business areas.

Figure 3 is a diagram that was developed at Oglethorpe in 1979. Showing only major functions and data bases, this type of diagram is a particularly valuable device for communicating with management and DP steering committee personnel. Although important in terms of a contextual picture, such diagrams are time-consuming to draw and maintain, and they tend to obscure important function and data relationships. Therefore, it may be better to develop more detailed models using Warnier/Orr charts (see Figure 4) for analysis of both functions and data. Such charts are useful because they:

- Can depict the significant business functions and data hierarchy
- Are reasonably easy to develop and maintain
- Communicate well with nontechnical personnel
- Provide an easier way to "explode" the models into more detail

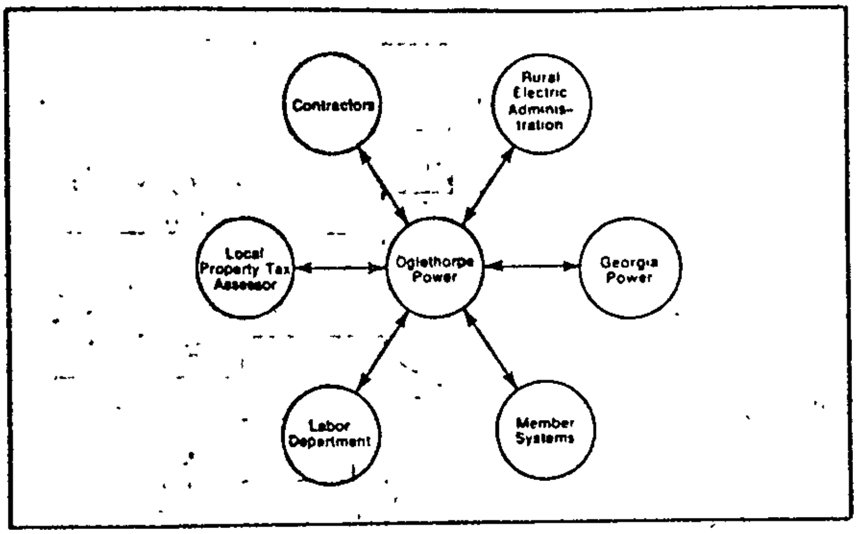


Figure 2. Simple Entity Diagram

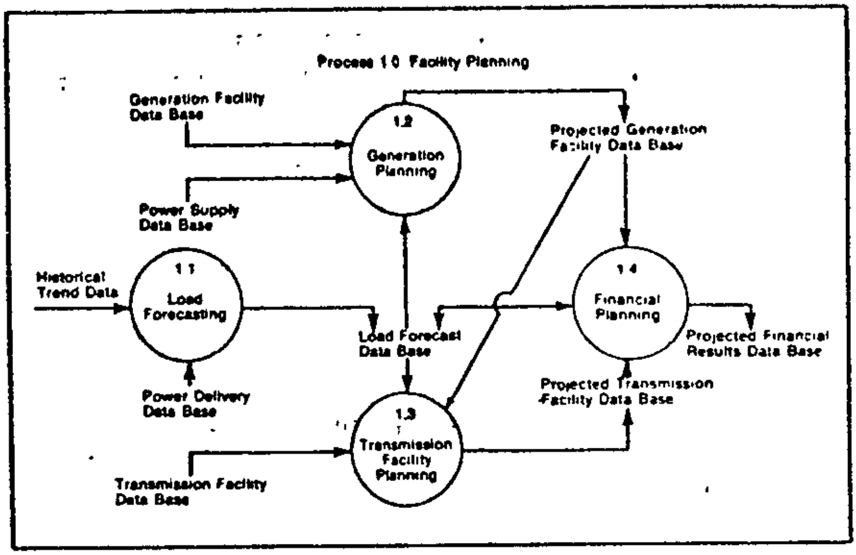

Process 1.0 Facility Planning
Generation Facility Data Base
1.2
Generation Planning
Projected Generation Facility Data Base
Power Supply Data Base
Historical Trend Data
1.1
Load Forecasting
1.4
Financial Planning
Load Forecast Data Base
Projected Financial Results Data Base
Projected Transmission Facility Data Base
Power Delivery Data Base
1.3
Transmission Facility Planning
Transmission Facility Data Base

Figure 3. Major Functions and Data Bases

The function model shown in Figure 4 was developed by analyzing the major objectives, responsibilities, functions, and tasks of each organizational unit and arranging them in a functional hierarchy. This model concentrates on high-level functional tasks, although further decomposition defining lower-level tasks is part of the next planning phase.

During the development of the function model(s), the raw data for the data models begins to accumulate. Although the usual sequence is

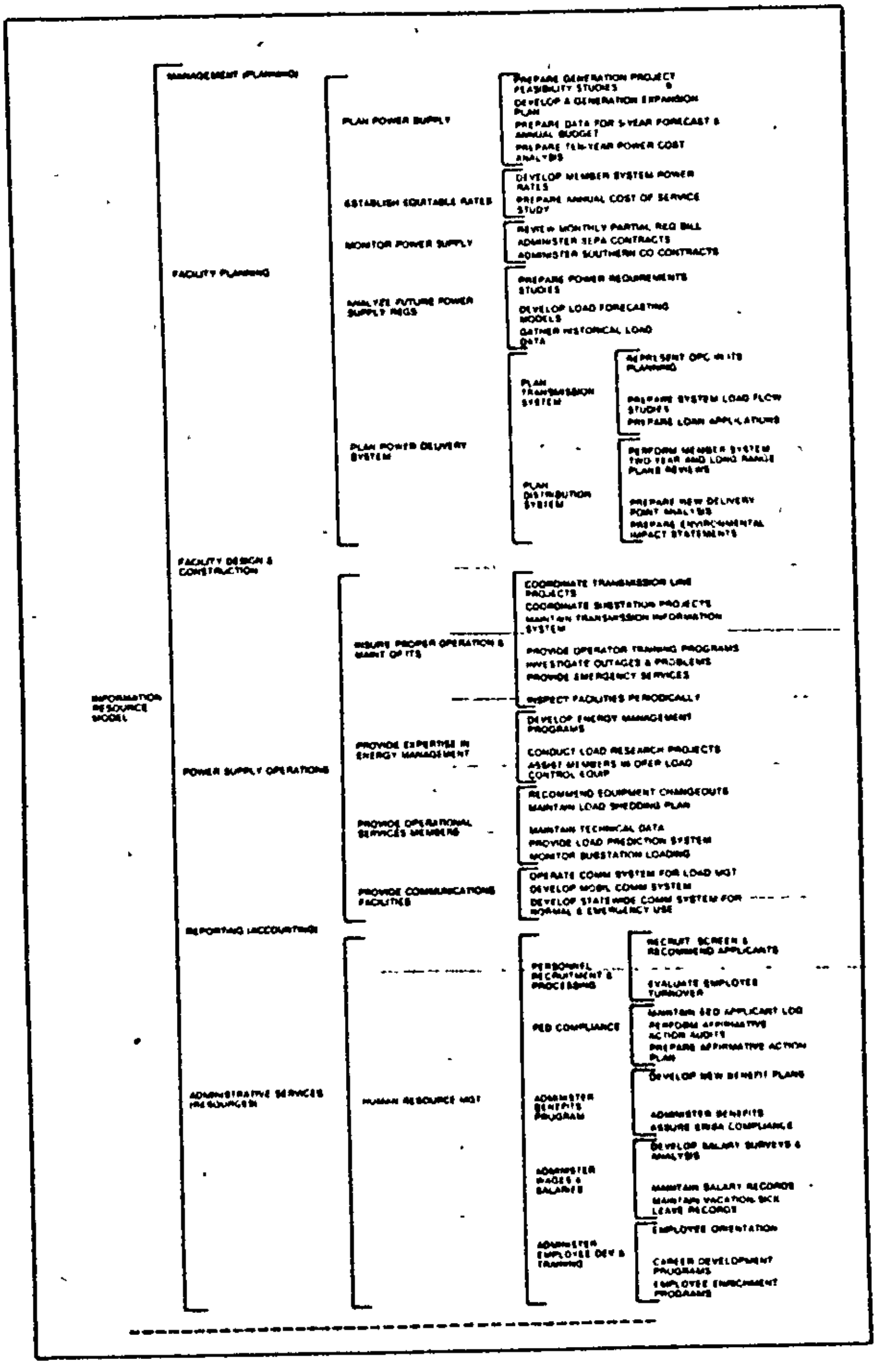

INFORMATION RESOURCE MODEL
MANAGEMENT (PLANNING)
FACILITY PLANNING
PLAN POWER SUPPLY
PREPARE GENERATION PROJECT FEASIBILITY STUDIES
DEVELOP A GENERATION EXPANSION PLAN
PREPARE DATA FOR 5-YEAR FORECAST & ANNUAL BUDGET
PREPARE TEN-YEAR POWER COST ANALYSIS
ESTABLISH EQUITABLE RATES
DEVELOP MEMBER SYSTEM POWER RATES
PREPARE ANNUAL COST OF SERVICE STUDY
MONITOR POWER SUPPLY
REVIEW MONTHLY PARTIAL REQ BILL
ADMINISTER SEPA CONTRACTS
ADMINISTER SOUTHERN CO CONTRACTS
ANALYZE FUTURE POWER SUPPLY REQS
PREPARE POWER REQUIREMENTS STUDIES
DEVELOP LOAD FORECASTING MODELS
GATHER HISTORICAL LOAD DATA
PLAN POWER DELIVERY SYSTEM
PLAN TRANSMISSION SYSTEM
REPRESENT OPC IN ITS PLANNING
PREPARE SYSTEM LOAD FLOW STUDIES
PREPARE LOAN APPLICATIONS
PLAN DISTRIBUTION SYSTEM
PERFORM MEMBER SYSTEM TWO YEAR AND LONG RANGE PLANS REVIEWS
PREPARE NEW DELIVERY POINT ANALYSIS
PREPARE ENVIRONMENTAL IMPACT STATEMENTS
FACILITY DESIGN & CONSTRUCTION
INSURE PROPER OPERATION & MAINT OF ITS
COORDINATE TRANSMISSION LINE PROJECTS
COORDINATE SUBSTATION PROJECTS
MAINTAIN TRANSMISSION INFORMATION SYSTEM
PROVIDE OPERATOR TRAINING PROGRAMS
INVESTIGATE OUTAGES & PROBLEMS
PROVIDE EMERGENCY SERVICES
INSPECT FACILITIES PERIODICALLY
POWER SUPPLY OPERATIONS
PROVIDE EXPERTISE IN ENERGY MANAGEMENT
DEVELOP ENERGY MANAGEMENT PROGRAMS
CONDUCT LOAD RESEARCH PROJECTS
ASSIST MEMBERS IN OPER LOAD CONTROL EQUIP
PROVIDE OPERATIONAL SERVICES MEMBERS
RECOMMEND EQUIPMENT CHANGEOUTS
MAINTAIN LOAD SHEDDING PLAN
MAINTAIN TECHNICAL DATA
PROVIDE LOAD PREDICTION SYSTEM
MONITOR SUBSTATION LOADING
PROVIDE COMMUNICATIONS FACILITIES
OPERATE COMM SYSTEM FOR LOAD MGT
DEVELOP MOBL COMM SYSTEM
DEVELOP STATEWIDE COMM SYSTEM FOR NORMAL & EMERGENCY USE
REPORTING (ACCOUNTING)
ADMINISTRATIVE SERVICES (RESOURCES)
HUMAN RESOURCE MGT
PERSONNEL RECRUITMENT & PROCESSING
RECRUIT, SCREEN & RECOMMEND APPLICANTS
EVALUATE EMPLOYEE TURNOVER
EEO COMPLIANCE
MAINTAIN EEO APPLICANT LOG
PERFORM AFFIRMATIVE ACTION AUDITS
PREPARE AFFIRMATIVE ACTION PLAN
ADMINISTER BENEFITS PROGRAM
DEVELOP NEW BENEFIT PLANS
ADMINISTER BENEFITS
ASSURE ERISA COMPLIANCE
ADMINISTER WAGES & SALARIES
DEVELOP SALARY SURVEYS & ANALYSIS
MAINTAIN SALARY RECORDS
MAINTAIN VACATION, SICK LEAVE RECORDS
ADMINISTER EMPLOYEE DEV & TRAINING
EMPLOYEE ORIENTATION
CAREER DEVELOPMENT PROGRAMS
EMPLOYEE ENRICHMENT PROGRAMS

Figure 4. High-Level Function Model

to analyze the functions first and then the information requirements, any analytical task—particularly planning—can be iterative.

As each function in the model is determined, the information required to perform that function and the resultant output or action are analyzed. This information forms the basis of the data models. The data to support higher-level managerial functions is usually unstructured and external to the organization; it should, therefore, be identified in broad generic terms. As the examined functions become more specific, the data analysis becomes more detailed.

As the information required to perform the identified functions is determined, data hierarchies begin to emerge. Unfortunately, multiple hierarchies emerge; since they often contain the same basic data elements, confusion and complexity soon reign.

One way to attack this confusion is to build network data models (see Figure 5); however, even though such data models may have good visual impact, they are difficult to decompose into more detailed elements. It is obvious that each box could itself contain a hierarchy of detail data.

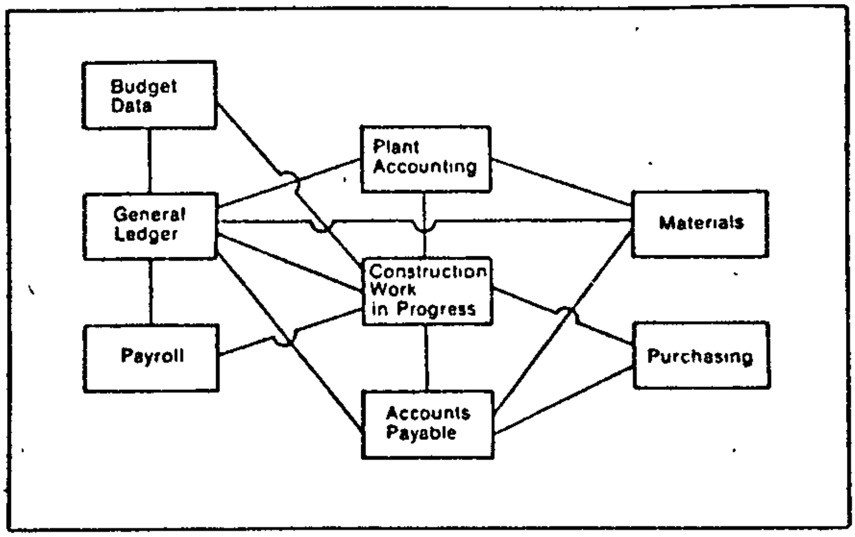


Figure 5. Network Data Hierarchy

In data analysis, network representations are difficult to understand except on a high level, and access paths between elements tend to be obscure. When hierarchical data structures instead of network structures are developed to support each major function, it is easier to communicate and therefore to incorporate more detail into the model. These data structures meet the dual objectives of communicating both business needs and technical requirements.

Figure 6 shows sample data structures that support a financial statement preparation function. Although this figure is uncomplicated, it

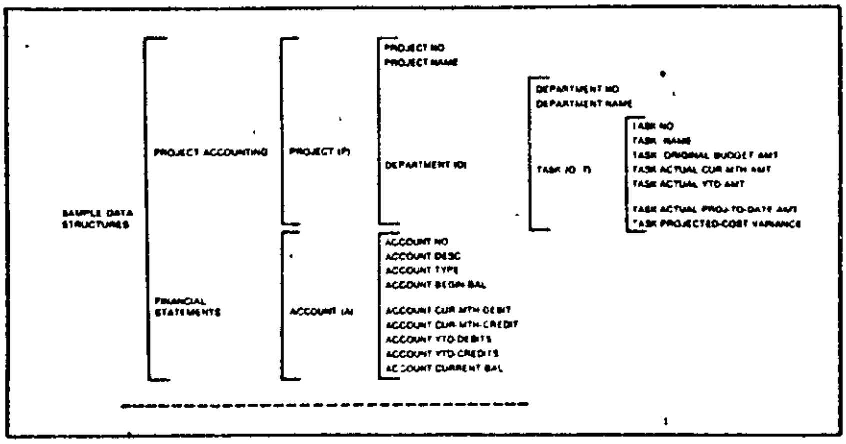


Figure 6. Sample Data Structures Supporting Financial Statement Preparation Function

conveys a significant amount of information. Project managers can analyze their data needs by examining one structure, accountants the other. It is then a technical task to combine structures into appropriate data bases.

Strategic Analysis Output Documents. The products produced during the strategic analysis described in this section are:

- A high-level network model showing the major business operations
- An entity diagram showing external relationships
- A function flow diagram showing major functions and data bases
- A function model showing a hierarchy of functions and subfunctions
- Data models that begin to identify and structure business information needs

Operational Models

The transition from the defined strategic analytical products to the definition of the right application project(s) which is the goal of the systems planning process, involves the following steps:

- Aggregation of functions into application group function models
- Definition of application group data models
- Analysis of function and data relationships—including development of a time-cycle chart
- Definition of application project function and data models

Functional areas that have some business relationship are combined into application groups. The strategic function model aggregates functions into a logical business hierarchy. The definition of application group models is a process of reaggregating these functions.

These groups can closely parallel the major functions identified in the overall systems network and provide a basis for further decomposition of the functions. The decomposition process has the same purpose as any other analytical task—to break a situation into components that can be managed conceptually.

An application group can be a financial group or an international banking group; an application system can be an accounts payable system or a letter of credit system. The number of levels and the scope of an application group depend on the size, type, and diversity of an organization. An application group can be developed for each division in a highly decentralized corporation or for an entire small corporation. Application groups are then further subdivided into individual, implementable application system projects.

Experience should be the starting point for developing application group models; a structured planning process does not start from scratch but provides tools that enhance prior knowledge. Application groups can include financial, marketing, manufacturing, facility planning, and administrative systems. These can be further divided into general ledger, accounts payable, sales forecasting, and order processing applications systems. Although this division may appear somewhat trivial, by allocating to these application groupings both the functions identified in the strategic function model and the data and information needs identified in the data models, the scope of individual application systems begins to become clear.

In the process of developing group and system models, two additional types of information must be incorporated: feedback and control mechanisms and function and data relationships.

Feedback and Control Mechanisms. Feedback and control are crucial to defining the requirements for a system:

> True systems are distinguished by their attributes. From a logical standpoint, true systems are: self-controlling, self-correcting, goal directed, and persistent . . . Management information systems can be developed empirically by surveying managers and asking them what they want to know, or they can be developed based on a scientific analysis of the business (functional) system it supports and the feedback/control required to ensure its successful operations. In the first case, the quality of the system developed is strictly limited by the skills and perceptions of a given set of managers and users at a specific time In the second case, *the quality of the system is based on the fundamental business functions*[6].

In systems planning, each of the major functions identified in the models should be examined for control requirements. For example, the control for various accounting functions may be an internal auditing function. In addition, the frequency with which functions and controls are performed may involve additional systems requirements. Figure 7 shows a time-cycle chart that can be used to visualize the timing or frequency of system activities.

Besides providing new function and data requirements, feedback and control analysis is also useful in setting priorities and establishing application system performance criteria.

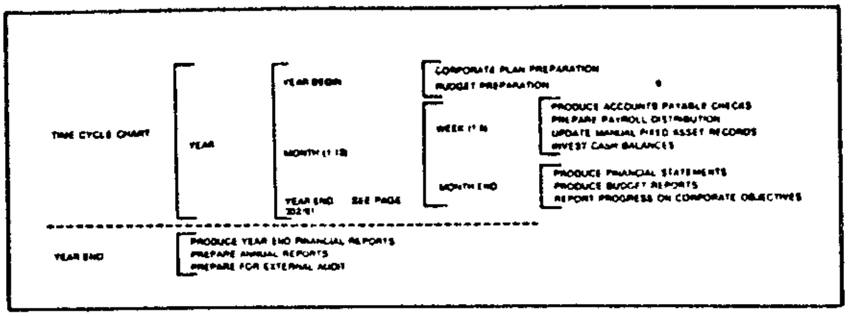

TIME CYCLE CHART
YEAR
YEAR BEGIN
CORPORATE PLAN PREPARATION
BUDGET PREPARATION
MONTH (1-12)
WEEK (1-N)
PRODUCE ACCOUNTS PAYABLE CHECKS
PREPARE PAYROLL DISTRIBUTION
UPDATE MANUAL FIXED ASSET RECORDS
INVEST CASH BALANCES
MONTH END
PRODUCE FINANCIAL STATEMENTS
PRODUCE BUDGET REPORTS
REPORT PROGRESS ON CORPORATE OBJECTIVES
YEAR END SEE PAGE
YEAR END
PRODUCE YEAR END FINANCIAL REPORTS
PREPARE ANNUAL REPORTS
PREPARE FOR EXTERNAL AUDIT

Figure 7. Time-Cycle Chart

Function Relationships. In addition to defining the scope of individual projects, the analysis of function and data relationships brings the sequence of implementation and the linkages to past and future systems into focus. Which functions should be automated first and the optimal implementation sequence become clear based on priorities expressed in the corporate plan and in projected implementation parameters (e.g., schedule and resource utilization).

Optimal implementation sequence is determined by analyzing the function flow (i.e., the sequence in which the functions should be performed). This function flow analysis can be done with high-level data flow diagrams or, as shown in Figure 8, a specialized Warnier/Orr chart called a functional flow chart.

Obviously, the priority sequence and the optimal implementation sequence often conflict. By recognizing this conflict during the planning process and assessing its impact, the project implementation sequence can balance these conflicting objectives. The greater the conflict between these two sequences, the more thorough and detailed the planning effort must be.

The scope of an application project, that is, the specific functions incorporated into that project, must also be determined. Conflicting objectives between projects of greater and lesser scope must be recognized and accommodated in the planning process.

Data Relationships. An analysis of the data relationships provides two types of information:

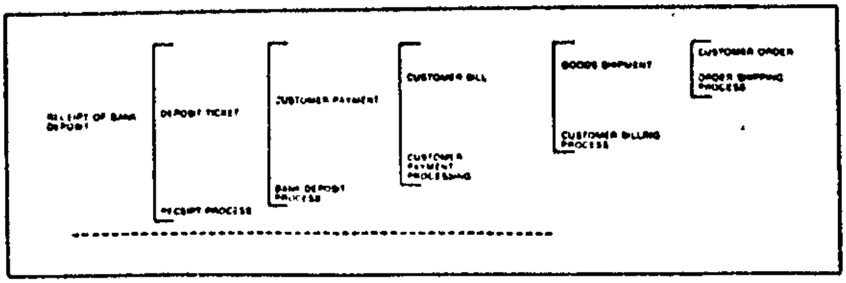


Figure 8. Functional Flow Chart

- An indication of a project's scope, since more data usually implies an expanded system
- The key linkage information between systems

Perhaps no other piece of planning information is as important, or as frequently missing, as that concerning linkages between current systems, those under development, and future systems. A brief example illustrates the point. If a payroll system is being designed as part of a financial application group implementation, the data hierarchy might look like the one in Figure 9. A project control system is also anticipated, but with low priority, so implementation is scheduled several years hence. Since the payroll system designers know the project control system is planned, they might allow for "Project Number" in their labor distribution design. The problem arises when the project control system design requires salary cost by project *and* task. If a data structure is developed for the project control system in a systems plan, specific linkages (access paths) can be better anticipated and allowances can be made in the design.

Scope (or boundary conditions) and linkages are directly related, because expanding the scope (functions) increases the information needs, which in turn changes the data linkage points (for a specific system). A good way to think of this is: as the number of functions included in the project scope increases, the number of data linkages decreases, and the implementation period approaches infinity.

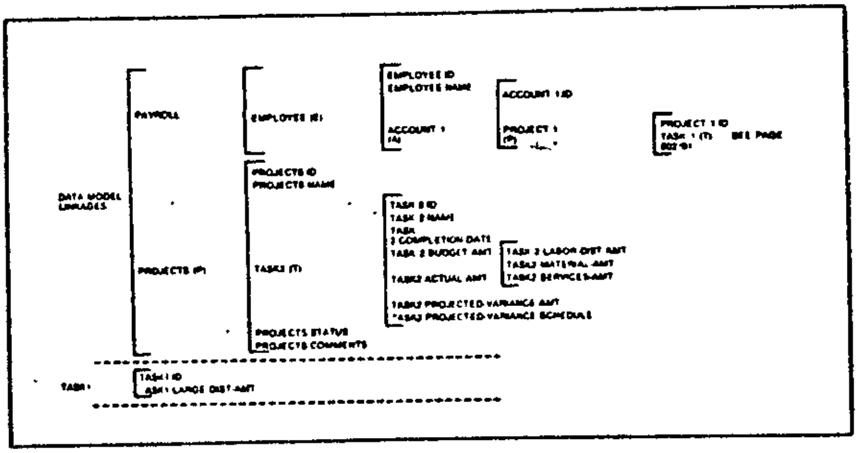


Figure 9. Data Model Linkages

Model Maintenance

Each application system project emerges from the planning process defined as a set of models. During each successive implementation stage—functional specification, technical design, installation, and operation—the system changes to reflect increased knowledge. Systems planning should be a continuous process. Although they are heavily emphasized only during the annual planning cycle, systems plans should not be rushed through once a year and then put on the shelf. The models should be used constantly as reference points during systems development and ongoing planning.

The degree of detail maintained in the planning models depends on the organization's needs. Because the key elements of the operations or project definition phase of planning are scope and linkages, these must be updated often so that subsequent planning does not proceed from an erroneous base. As more detailed planning and development work is done, the information about functions and data may show that a restructuring of the high levels is required.

EVOLUTION OF SYSTEMS PLANNING AT OGLETHORPE: A CASE STUDY

This section contains a brief chronology of the evolution of the planning process at Oglethorpe Power Corporation. Systems planning at Oglethorpe begins with the annual corporate business plan, a five-year plan that concentrates on the upcoming year. In 1979 and 1980, much effort was put into more closely integrating the corporate and systems planning processes.

After the managers develop their business objectives, the DP requirements to assist in meeting those objectives are identified. The priority of DP projects then closely parallels corporate priorities. The DP steering committee sets priority, scheduling, and personnel levels based on DP's estimate for each project.

Past Methodology

During the 1979 planning cycle, management decided that a framework to aid communications between user department heads and the steering committee and to provide DP with a plan that transcended individual application systems was needed. The result was the Information Resource Model shown in Figure 1. The model was constructed from IBM's Business Systems Planning modified data flow diagrams [3] and very brief high-level data base descriptions. It was developed for these reasons:

- To facilitate communications between the DP and client departments
- To offer another perspective on organizational relationships
- To assist in determining systems development priorities
- To assist in determining the optimal development sequence
- To facilitate development of multi-use data bases
- To provide a plan for linking systems together

Requirements for the model included minimizing technical terminology, maximizing visual impact, and providing a tool to be used at all organizational levels.

The information resource model, the more detailed functional flow models, and the brief data base descriptions became the framework on which individual projects were based. This framework proved useful throughout 1979 during development projects and in communicating with users about the context of new systems ideas.

Enhancements to the 1979 Process

During 1979 Oglethorpe undertook a study to determine which of the various structured development methodologies to implement. It was decided to:

- Implement Warnier/Orr structured development techniques
- Install Method/1 Systems Development Practices
- Install Langston, Kitch and Associates' STRUCTURE(S) software package as a structured documentation tool

These tools enhanced the 1980 planning process.

The primary systems planning focus in 1980 was to complete an extensive application group plan for the development of corporate financial systems. Extensive requirements in this area for the next 14 years dictated a careful approach to establish the system's proper scope and avoid duplicate effort, conflicting data, and systems rework.

Output from this plan included:

- A financial application group function model
- Financial application group data models
- A financial application group time-cycle model
- Identification, description, and prioritization of systems projects
- General requirements for all systems in this systems group
- Implementation alternatives including overall software strategy

Structured techniques and tools were used, where appropriate, to complete this plan. The function and data models were illustrated with Warnier/Orr charts (see Figure 10), and entity or context diagrams showed major corporate interfaces with other organizations and interfaces among internal departments (see Figure 11). The STRUCTURE(S) package facilitated modification of the models during the planning stage, provided permanent documentation, and will allow maintenance of the models as actual projects are undertaken.

The systems planning and functional specification sections of Method/1 identified and organized the tasks for consideration; these practices were supplemented with structured techniques for actually accomplishing the tasks.

During the 1980 planning process, Oglethorpe began developing an overall function model for the organization, although time constraints limited detail outside the financial area.

Future Plans

Future plans call for refining the techniques currently in use and expanding the models while increasing the level of detail. The optimum amount of detail will be determined by experience. Balancing the need for sufficient detail for adequate planning against the accumulation of

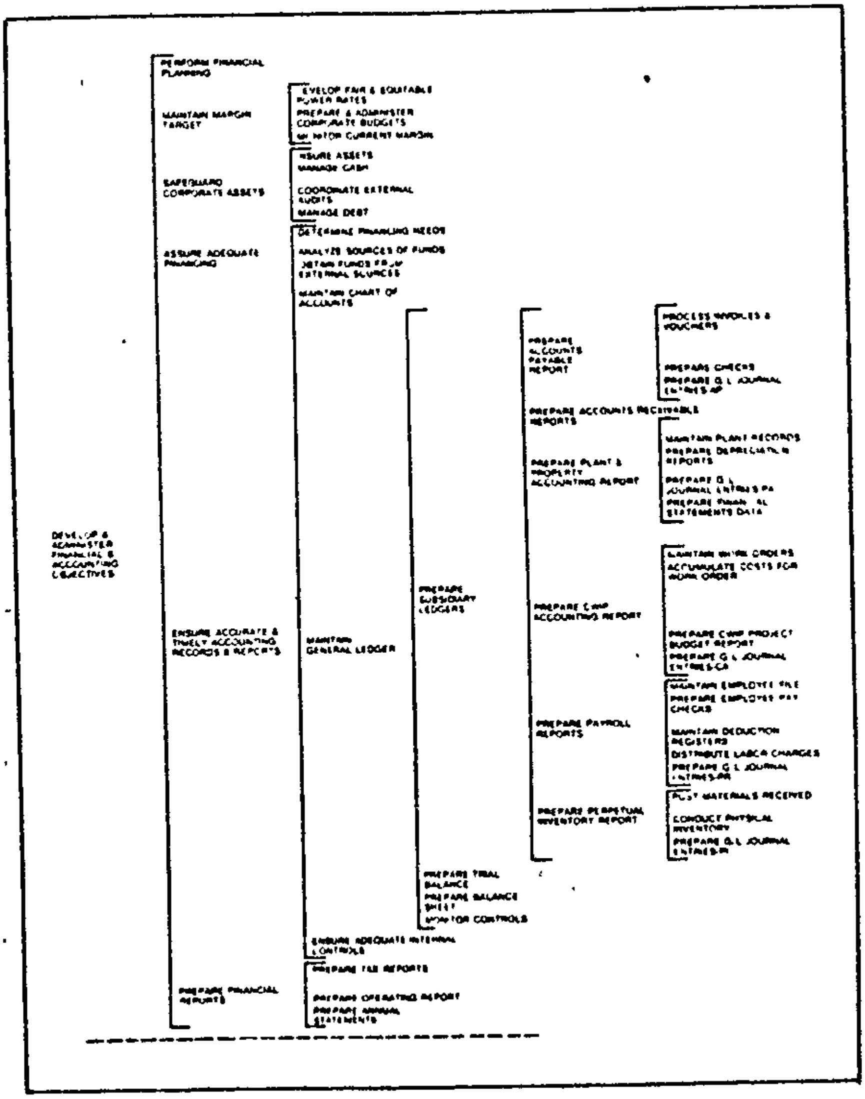

DEVELOP & ADMINISTER FINANCIAL & ACCOUNTING OBJECTIVES
PERFORM FINANCIAL PLANNING
MAINTAIN MARGIN TARGET
DEVELOP FAIR & EQUITABLE POWER RATES
PREPARE & ADMINISTER CORPORATE BUDGETS
MONITOR CURRENT MARGIN
SAFEGUARD CORPORATE ASSETS
INSURE ASSETS
MANAGE CASH
COORDINATE EXTERNAL AUDITS
MANAGE DEBT
ASSURE ADEQUATE FINANCING
DETERMINE FINANCING NEEDS
ANALYZE SOURCES OF FUNDS
OBTAIN FUNDS FROM EXTERNAL SOURCES
MAINTAIN CHART OF ACCOUNTS
ENSURE ACCURATE & TIMELY ACCOUNTING RECORDS & REPORTS
MAINTAIN GENERAL LEDGER
PREPARE SUBSIDIARY LEDGERS
PREPARE ACCOUNTS PAYABLE REPORT
PROCESS INVOICES & VOUCHERS
PREPARE CHECKS
PREPARE G L JOURNAL ENTRIES-AP
PREPARE ACCOUNTS RECEIVABLE REPORTS
PREPARE PLANT & PROPERTY ACCOUNTING REPORT
MAINTAIN PLANT RECORDS
PREPARE DEPRECIATION REPORTS
PREPARE G L JOURNAL ENTRIES-PA
PREPARE FINANCIAL STATEMENTS DATA
PREPARE CWP ACCOUNTING REPORT
MAINTAIN WORK ORDERS
ACCUMULATE COSTS FOR WORK ORDER
PREPARE CWP PROJECT BUDGET REPORT
PREPARE G L JOURNAL ENTRIES-CA
PREPARE PAYROLL REPORTS
MAINTAIN EMPLOYEE FILE
PREPARE EMPLOYEE PAY CHECKS
MAINTAIN DEDUCTION REGISTERS
DISTRIBUTE LABOR CHARGES
PREPARE G L JOURNAL ENTRIES-PR
PREPARE PERPETUAL INVENTORY REPORT
POST MATERIALS RECEIVED
CONDUCT PHYSICAL INVENTORY
PREPARE G L JOURNAL ENTRIES-PI
PREPARE TRIAL BALANCE
PREPARE BALANCE SHEET
MONITOR CONTROLS
ENSURE ADEQUATE INTERNAL CONTROLS
PREPARE FINANCIAL REPORTS
PREPARE TAX REPORTS
PREPARE OPERATING REPORT
PREPARE ANNUAL STATEMENTS

Figure 10. Financial Plan Functional Model

excessive detail, which can hamper analysis, is a continuing effort. Note that as these models are developed, they become increasingly important to application system specification and design.

RECOMMENDED COURSE OF ACTION

It is presumptuous to draw hard and fast conclusions in the somewhat ethereal realm of planning, particularly systems planning. Information systems literature abounds with predictions of future trends in hardware, software, office automation, the demise of programmers, and the rise of user-friendly systems. Although some of the following

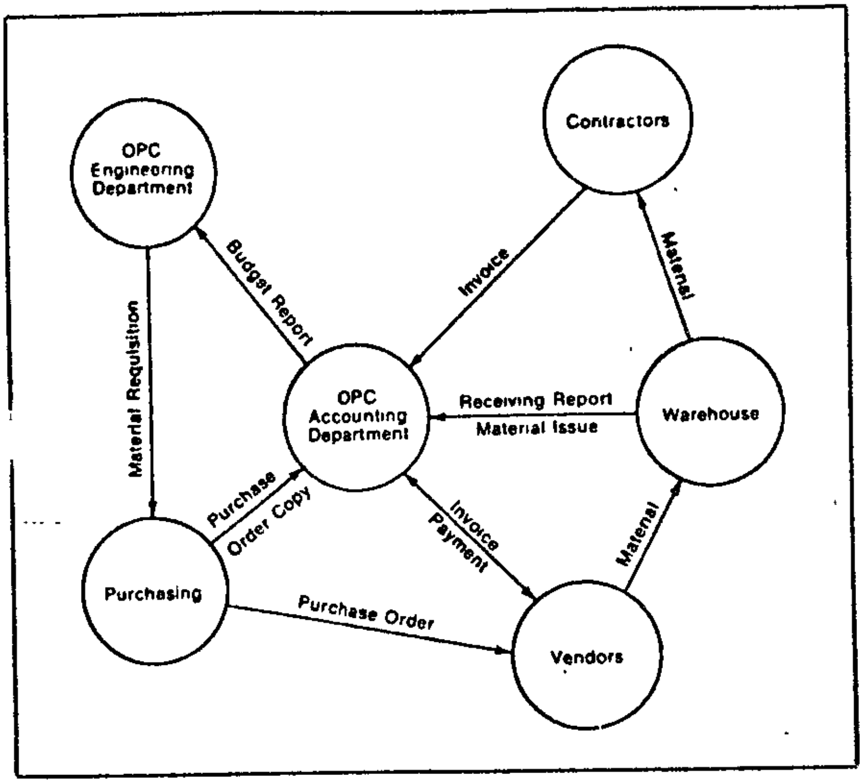


Figure 11. Financial Systems Plan: Entity Diagram for Construction Work in Progress

statements may sound like those predictions, they are actually reflections on experience at Oglethorpe.

Since some of the techniques discussed in this portfolio are in the experimental stages of development and use at Oglethorpe, quantifiable results are lacking. The intangible results, however, point toward a better vehicle for communication with user management and the DP steering committee and toward a better definition of systems projects.

The benefits of structured syst[illegible] planning, therefore, are potentially significant. Realizing the[illegible]s requires the development of a viable, detailed, and structure[illegible]al plan that bridges the gap between strategic DP planning and the initiation of specific application systems projects.

The risk of future integration problems increases in direct proportion to the decrease in the scope of each development phase, particularly the planning phase This risk and the overall development time can be reduced by the development and maintenance of detailed function and data models as part of operational planning and implementation processes. Whether structured systems planning can assist DP organizations in bringing increased technical sophistication to bear more effectively on corporate objectives is a question that remains to be answered.

This portfolio was written by Jim Highsmith, Ken Orr and Associates, Topeka KS, formerly of Oglethorpe Power Company, Atlanta GA. Reprinted by permission from "Structured Systems Planning," by Jim Highsmith, *MIS Quarterly* Volume 5, Number 3 (September 1981). Copyright 1981 by the Society for Management Information Systems and the Management Information Systems Research Center.

References

1 Arthur Anderson & Co. *Method/1: Systems Development Practices*. Chicago, 1979.

2 Blumenthal, S.C. *Management Information Systems: A Framework for Planning and Development*. Englewood Cliffs NJ: Prentice-Hall Inc., 1969

3 *Business Systems Planning—Information Planning Guide*, GE20-0527-1, White Plains NY: IBM Corporation, 1975

4. Drucker, P.F. *Management: Tasks, Responsibilities, Practices*. New York: Harper & Row, 1973

5 King, W.R. "Strategic Planning for Management Information Systems." *MIS Quarterly*, Vol. 2, No. 1 (March 1978).

6. Orr, K.T. *Structured Systems Development*. New York: Yourdon, Inc.

7 Orr, K.T. "Structured Requirements Definition." *Software News*, Vol. [illegible] No. [illegible] ([illegible] 1981), 31

8. Zachman, J.A. "The Information Systems Management System: A Framework for Information Systems Planning." Society for Management Information Systems *Proceedings of the Ninth Annual Conference*, 1977.

INSIDE:
Characteristics of Information, Analysis of Present and Projected System, Cash Flow, Cost Summaries

(REV 4/77) 32-03 04

INFORMATION SYSTEMS ANALYSIS

Selecting and Analyzing Alternatives

Performing Cost/Benefit Analysis

PAYOFF IDEA. Like political rhetoric after the election, the Cost/Benefit Analysis (CBA) may be forgotten after the project implementation begins. However, it is an extremely important document because it has been the vehicle by which management approval was obtained. It is time for systems development personnel to be serious in meeting the projections set forth in the CBA, both to justify management's confidence and to meet personal goals of systems implementation.

PROBLEMS ADDRESSED

There is probably no more influential segment in a feasibility study for a potential application than the cost/benefit analysis. Indeed, in most instances the future of the proposed project will be decided by the results of this analysis. Once technical feasibility has been assured, management's decision to abort or proceed with the project will usually be made on the same basis as with any other investment — the adequacy of the return on the investment as compared with other demands for the organization's funds.

This portfolio addresses three topics that provide a background to C/B analysis: the value of information, the analysis of costs, and the analysis of benefits.

Before proceeding, a word of caution is necessary. Cost/benefit analyses cost money and time. Naturally, the more that is spent, the more refined will be the analysis, and (hopefully) the more accurate will be the conclusions An extensive study can reduce the uncertainty of the conclusions. As in considering any other investment, one would not wish to spend money out of proportion to the expected return. On a small project, or one for which the payoff is fairly obvious, it would be wasteful to perform a comprehensive analysis

DETERMINING THE VALUE OF INFORMATION

In addressing the value of information, the authors Gregory and Van Horn have focused upon such elements as accuracy, quantity, timeliness, relevance, and the consequences of information.[1] All of these might be summarized in the term "quality," which can be conceptualized as the balance between value and cost.

Cost and Quality

One might initially assume that the value of information increases in direct proportion to the quality. But there are some cases in which this is not true. For example, if the quality is improved by speed, does it pay to implement an on-line fixed assets depreciation system with inquiry capability when depreciation is reported monthly? This extreme example shows that, beyond a certain point, the value of information becomes less responsive to increases in quality. (See Figure 1.)

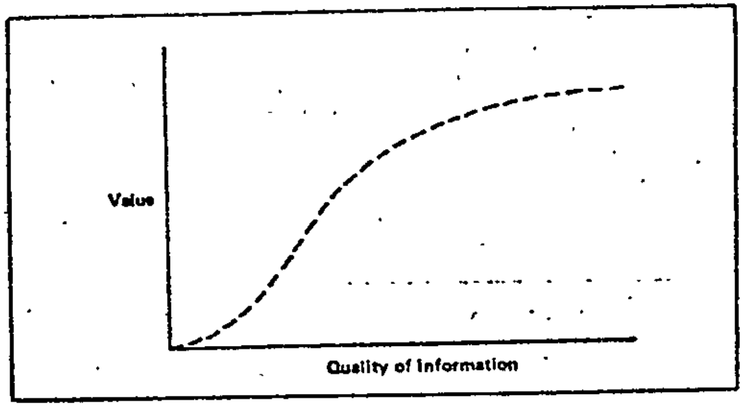


Figure 1. Value as a Function of Information Quality

Similarly, the relationship of quality and cost is not constant. At the lower level, relatively small investments can gain substantial improvements in information quality. In complex, highly sophisticated systems, on the other hand, large sums spent on further improvement may result in relatively small increments of quality gained.

Each level of quality represents a different set of specifications. Given an available technology, the most efficient set of specifications would produce a curve such as that shown in Figure 2. Information is obtained from a specific system. Alternative systems vary in their efficiency, and so the cost of a given quality of information depends on the efficiency of the design used. The *efficiency frontier* represents the set of systems that provide each level of quality at the lowest cost. For a given level of efficiency, cost rises with increased quality.

These curves are, of course, generalizations It would be difficult to assure the validity of such curves based on estimates for a proposed application. However, they do serve to illustrate the concepts.

These curves comparing value to quality, and cost to quality, are superimposed in Figure 3 to find the range of optimum quality. The point where incremental value matches incremental cost is also the point identifying the design configuration for a system that maximizes net benefits. Obviously, the optimum system does not provide all useful information; there will always remain unfulfilled information "requirements" that cost more to satisfy than they contribute in benefits.

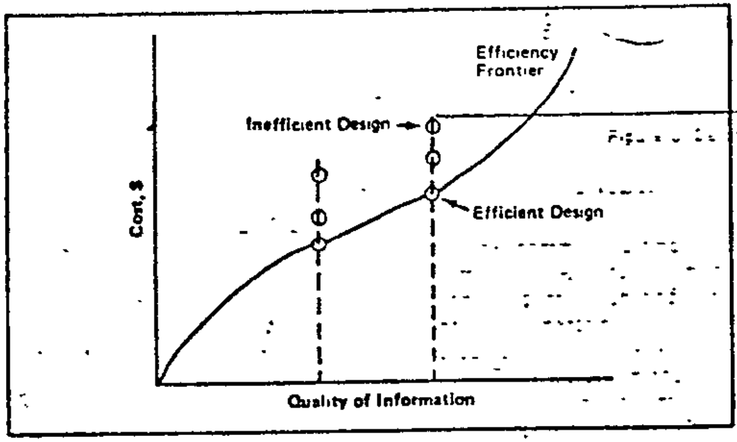


Figure 2. Cost as a Function of Information Quality

Cost and Technological Advances[2]

Consideration should also be given to the fact that costs are responsive to advances in technology. Experience over the last 20 years of data processing shows that each generation of computers has repeatedly reduced the cost per element of data processed. Both hardware and software advances have not only lowered the cost of information processing, but have provided new levels of capability and capacity.

Responses to the advances of technology can take several forms In the simplest approach, some organizations may take advantage of new technology to lower the cost of processing only. This could be done by a straightforward conversion of the present system to the new technology. In terms of our cost/value concept, this would shift the cost curve to the right, leaving the value curve in place (assuming that, since no changes are made to the system, the value of the information remains constant).

On the other hand, the system could be redesigned to take advantage of new capabilities provided by the new technology This might enhance the information value as well as reduce the cost of processing The resulting optimum system design (the point at which net benefits are maximized) would be at a higher level of quality.

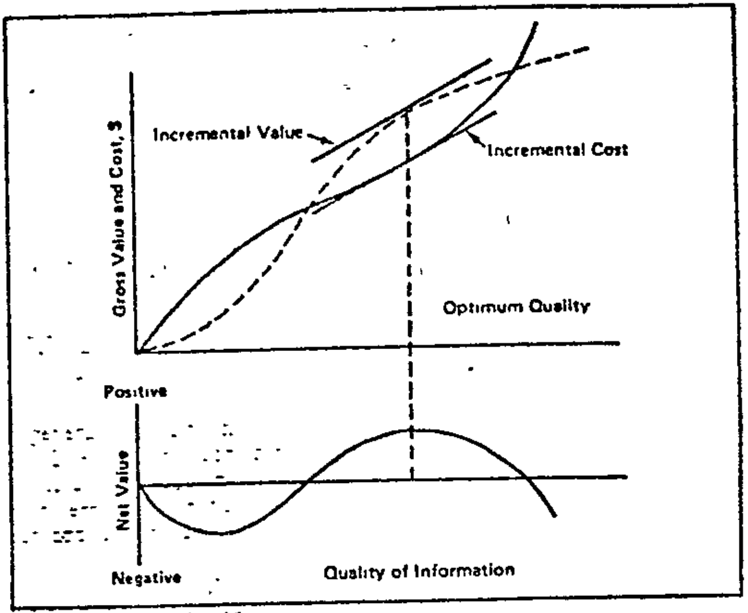


Figure 3. Determining the Optimum System

Accuracy of Information

One common justification of DP systems has been accuracy of information. Accuracy cannot be viewed as a binary function; it has various degrees. In addition, increasing accuracy implies increasing the cost of operating a system. These increased costs appear in such areas as:

- Error detection routines in programs.
- Collection and maintenance of redundant information for error detection and correction.
- Externally maintained controls.
- Hardware devices for increasing input accuracy.
- Internal program-to-program run controls.
- Redundant processing
- Extensive program testing and validation
- Rigid and extensive program-change control procedures.

The amount spent on each of these activities should be consistent with the level of accuracy demanded by the system. Accuracy has different values for different purposes.

Establishing a program for computing results to three decimal places is futile if the input data is reliable only to one decimal place. It could be dangerously misleading if the user assumes the validity of the apparent level of accuracy. Available input reliability should dictate the level of accuracy designed into the processing functions. In no case should reports imply an accuracy that does not exist.

Accuracy, therefore, ideally consists of presenting the closest possible picture of reality. Decisions based upon information presented to a user may be more or less optimal, based on how closely that information conforms to reality.

Organizationally, the operating activities generally require higher levels of accuracy than administrative activities. One need only consider the difference between a few dollars in the wrong account on a month-end closing statement and that same amount of error on a paycheck. High levels of accuracy are demanded by such applications as payroll, check reconciliation, purchasing, invoicing, etc

A certain tolerance for error is expected in some systems such as inventory, shop floor control, and those for which periodic verification, such as physical inventory or replanning of the work flow, is a part of the ongoing system and results in regular replacement or validation of the files.

Another level of accuracy is permitted in monthly reports to top management which are often summarized and rounded to the nearest thousand dollars. Still another level should be anticipated for demographic studies such as market surveys, opinion polls, and censuses.

The rule for accuracy, then, is that *the cost of operating a system will be unnecessarily increased by specifying levels of accuracy beyond those actually valid or usable.*

Quantity of Information

It seems obvious that the cost of processing data correlates to the volume of data processed. But an examination of the reasons for large quantities of data may also provide avenues for reducing both volume and cost.

Again, let us consider the value of information This value is directly proportional to its "surprise" content. That is, information that you already know is categorized as redundant and worthless; information that confirms expectations has increasing value, but information that illuminates the unexpected or previously unknown has the highest possible value Of course, the underlying assumption here is that the information is relevant to the operational and decision-making needs of the firm.

Timeliness of Information

The element of speed in data processing has two aspects. The first is the speed or frequency of updating the information, and the second is the speed of response to user needs or the frequency of reporting It would be costly to assume that these two are automatically linked. Applications may require quick update and retrieval, quick retrieval only, or neither In some applications, the need for on-line update and response is readily justifiable Examples include airline reservation systems, process control systems, etc

Some inventory and production control applications dealing with very high volumes may justify on-line update and response in terms of avoiding stock-outs while minimizing the size of the inventory carried In many such applications it is sufficient to provide overnight updating with

on-line inquiry or overnight updating and reporting. Speed of response dictates that the processing method chosen may vary in cost. Tape-oriented batch processing may be least costly, but with a demand for faster response time and an increased frequency of processing, this method can become burdensome and costly. The next step would be disc-oriented indexed sequential processing Finally, if response time is reduced to the minimum (immediate, or on-line inquiry), a random processing approach becomes most economical. The comparison of the costs of these methods is illustrated by Figure 4.[5]

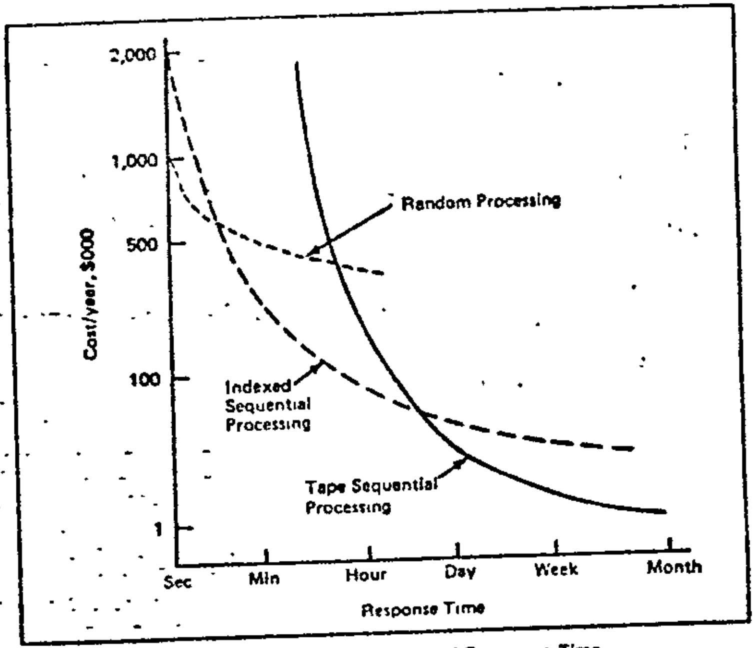


Figure 4. Cost as a Function of Response Time

Extent of Automating Data Processing

A final consideration in the cost/value trade-off is to determine how much of an application should be automated An old marketing observation is that 80 percent of the business is done with 20 percent of the clientele. This "80/20" rule has been found to be applicable to many other areas.

In application design, we may find that 80 percent of the benefit can be gained from spending 20 percent of the cost, while to obtain the remaining 20 percent of benefit will require 80 percent of the project cost The proportions, in practice, may be different, but the rule carries the sense that automation should stop at a point within the real areas of payoff.

Generally, simple decision-making that follows clearly defined and fairly constant parameters can be automated. Complex decision-making in an area of flexible and ill-defined goals is the forte of humans

Each process in the proposed application should be examined for allocation to either machine or human processing The only reason to include a task that could be done more efficiently by a human in the automated portion of a system is if the results of that task are integral to the further processing of information by the total application.

ANALYSIS OF COSTS

Analysis of costs contains three major elements, which will be examined in the following section

- Present operating costs and their projection.
- Operating costs of proposed alternatives.
- Implementation costs of the alternative system.

There are two overall considerations regarding cost/benefit analysis that should be noted before proceeding Since the decision to develop and implement a new system involves a one-time investment, the analysis of costs and benefits must first be projected over sufficient time into the future to amortize the initial investment Corporate investment guidelines can usually be obtained from the Accounting Department. These guidelines may indicate that corporate management expects a payback on investments in 1, 2, 3, or 5 years (in some cases even longer). With this guideline, the projection should not be carried much further than the stated payback objective since any proposal that does not meet the criteria within that period would be economically unfeasible.

One other guideline should limit projections — the anticipated life of the system If the anticipated life of the new system is estimated at 3 years (for example), after which replacement or major modification would be required, this period should limit the projection regardless of payback criteria Anticipated minor modifications within the life of the system may simply appear as costs within the projection

The second overall consideration is the type of cost to be used — fixed and/or marginal Generally, fixed costs may be defined as those that do not vary with volume, while marginal costs are those that change as volume changes.

Fixed costs may include such items as building rent, property taxes, business licenses. If they remain fairly constant despite volume changes, some other areas are often included These may be power supply, telephone, and even such functional areas as purchasing, inventory control, payroll operating expense, and management.

Marginal costs would include raw materials, labor, supplies, tools, etc. If a fixed level of personnel is required regardless of volume, then only that portion that would change due to the volume of work would be correctly considered to be marginal.

Performing an analysis using both fixed and marginal costs leads to an unnecessary amount of work. Therefore, unless the organization's management requires the analysis to contain both, the analyst is well advised to simplify the task.

Since the objective of a new or modified application is often to affect the level of fixed cost, the firm's usual definitions cannot be used in the analysis Instead, the analyst must carefully identify those cost elements that will be impacted by the new application, defining all others as fixed cost. A statement of fixed cost elements should be contained in the analysis, specifically excluding them from consideration.

Cost of the Present System

The cost elements described in the following are treated as marginal costs unless otherwise identified Restricting fixed cost elements to those that are positively identified provides the widest latitude for exploring cost/benefit

Operating Personnel. Staff costs frequently represent the major expense of the existing system Using the documented analysis of the existing system that should have been completed prior to this phase of the study, the activities of all personnel in the user area can be readily identified Those people involved in activities relating to the proposed application form the operating personnel base. If administrative personnel are directly involved in the application and if their numbers may be affected by the change, they should also be included. Similarly, personnel who work part-time on the application should be included to the extent of the time dedicated.

If possible, avoid using average rates for personnel, and use actual salaries or hourly rates. In the benefit analysis, the user will be asked to identify personnel savings and these should be consistent with the costs stated.

At a detailed level, it may be wise to identify personnel costs by function within the application, since the new system may affect each functional area in a different manner

Overhead. Floor space, utilities, telephone costs, etc. should be included only if they may change as a result of the new system. Otherwise they should be classed as fixed costs. Employee benefits, however, should always be considered a part of the marginal cost. A rate for application to base salaries is used by most companies for budgeting employee benefits and can be obtained from the Accounting Department.

Computer Costs. If the existing system is an automated system, computer costs must be included Several considerations make this a complex area to estimate. First, if the new application will not create a need for additional computer hardware or operating personnel, is it reasonable to consider this cost as marginal? In this event, should supplies be the only computer-related marginal cost?

Second, although the new application may not, of itself, force the acquisition of new equipment, it may hasten the time when the total workload exceeds capacity As a result, some other, unrelated application may require an unwarranted amount of justification

The cost analysis should address all of these The elements of DP cost that will change should be identified and shown as a part of the marginal cost. These could include supplies, data entry personnel, data entry equipment, and data storage media. If an extra shift of computer operators must be added, this cost may be included.

The cost in terms of computer capacity should be recognized in a separate section from the computation of marginal cost If an internal rate exists for computer time, this may be multiplied by the time used for the application. If no internal rate is used, the time utilized by the application should be noted and later compared with the time utilization expected by the proposed alternative methods This approach will permit management to judge the alternative methods of allocating the existing computer resource

Included in this statement should be a measure of the limitation of the computer resource The computer has a relatively fixed cost that is responsive to volume changes in fixed increments. Exceeding computer capacity by a small amount may trigger a substantial increase in cost to obtain the next incremental level of capacity. Therefore, management's clear understanding can only be obtained by showing the present level of computer use (either as a percent of total available time or as a number of hours per month or day compared with the hours available for the same period). This would be compared with the level anticipated by the proposed application.

Maintenance Costs. This is another factor of existing system costs and may relate to the maintenance of office equipment or facilities. Maintenance costs may also include the systems and programming effort devoted to maintaining the effectiveness of the existing system. Computer time for this activity should also be included on the same basis as run time

Supplies. This cost may include special forms, cards, magnetic tape, and similar computer or general office supplies If available accounting records do not permit analysis for the particular application, estimates of use may be obtained from user personnel, verified by user management, and priced by consultation with the Purchasing Department as to average prices or by reference to the company's "internal supplies catalog."

Amortization. If the existing system is automated and the firm has a policy of capitalizing system development, has the existing system been amortized? If not, what portion of the capitalized cost must be written off as expense at the time the new system is implemented? (Even without a policy of capitalizing such expenses, management may want to know how effective this application area has been in achieving its previous goals.)

Office equipment and other capitalized items may carry amortization charges to the application area. These charges should be considered if equipment will be eliminated or changed. Note that these charges will not be eliminated if the company elects to warehouse the equipment for an indefinite period But they will be eliminated if the equipment can be used within a reasonable time to avoid the purchase of such equipment in another area, or if sold

Projecting the Present System's Costs

As previously indicated, the costs of the present system should be projected for sufficient time to cover the life expectancy of the new system or the payback period, whichever is least.

In some instances, the expected increase in cost due to continued use of the present system compared with its replacement would be most dramatically demonstrated over a longer period. Similarly, when the benefits obtained from replacement would be most significant near the end of the payback period, it may be appropriate to extend the time period analyzed.

The following factors should be considered when projecting the costs of the existing system:

Operating Personnel

- Impact of increases or decreases in business volume on number and type of personnel.
- Impact of regular salary increases and inflation.
- Impact of labor market for the type of personnel required, and consequent salary ranges.

Overhead

- Potential increases in employee benefits resulting from inflation, and existing or pending legislation.
- Changes resulting in increases or decreases to floor space, utilities, telephones, etc. from increases or decreases in business volume.

Computer Costs

- Inflation.
- Impact of increases of transaction volume on computer utilization and equipment requirements.
- Increases or decreases of marginal cost as affected by changes in volume.

Maintenance

- Increasing cost of maintenance as a function of system age.
- Increasing cost of equipment maintenance due to inflation and age.
- Cost of replacing worn-out equipment.

Supplies

- Inflation.

Amortization

- Normally will not change unless equipment must be replaced, added to, or sold as a result of obsolescence or increases or decreases in business volume,

Other Considerations

- Business volume may be constrained by the nature of the existing system. This may either be shown as a cost of continuing the present system or as a benefit of the new system.

A key factor in making projections is the recognition that business volume may either increase or decrease in the future. The author has witnessed decisions to implement new systems in two different situations that resulted in substantial cost to the companies involved when business volume declined and the cost of the new systems was too high. This is the result of conversion to systems with a higher fixed cost base than previously existed

By projecting costs for both likely increases and likely decreases in business volume, this potential trap can be avoided.

Projecting Proposed Alternative Systems' Costs

Projecting costs on the basis of proposed functional specifications for a system is, to a great extent, a guessing game. It is a challenge to the analyst to arrive at the best possible guesses for the expected operating costs. Yet, in order to support the best possible managerial decision, every attempt must be made to reduce the unknowns and the risk in the decision-making process.

The elements of operating cost to be estimated are substantially the same as those determined for the existing system. Differences, if any, are noted in the following

Operating Personnel

- Methods of calculation are the same.
- Adjust headcount for personnel who will be terminated or transferred from the application area.
- Adjust for changes in the type of skills or the job classifications of personnel who will be needed to operate the new system.

Overhead

- Calculations are the same.

Computer Costs

- Calculation methods are the same.
- Include any additional peripherals or special equipment that will be dedicated to the new system. For example, if normal growth would require the addition of two disc drives during the next year, but four will be required as a consequence of implementing the new system, then it is reasonable to attribute the cost of the two drives to the new system
- Follow company practice in capitalizing items over the defined dollar value The amortization of these purchased items is shown as operating cost for the system.
- Expense items acquired during the implementation period may be charged to the cost of development and implementation. Expense items acquired later should be shown as operating costs.

Maintenance

- Even a new system will require maintenance. This point should be stressed to user management to avoid disappointment. Maintenance during the first 6 months of a major system may be considered as either operating cost or a part of implementation cost. Thereafter, anticipated maintenance costs should be treated as operating cost.
- Calculation methods are the same

Supplies

- Calculations are the same.

Amortization

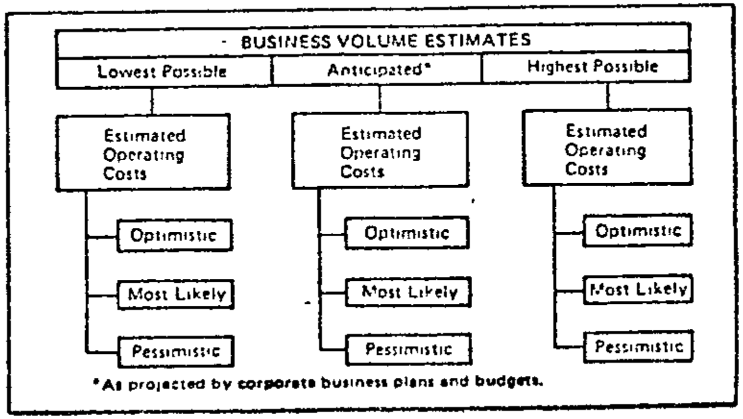
- Calculation methods for new equipment are the same.
- Even if corporate policy does not require capitalization of system implementation cost, a valid cost/benefit study demands such figures in the comparison The implementation cost of the new system should be amortized over the expected life of the system and shown as operating cost.

Other Considerations

- The new system may provide capabilities that did not previously exist The value of such capabilities can be estimated by user management in terms of their contribution to net profit before taxes and shown as earnings resulting from the new system.

In addition to a simple consideration of the preceding elements of cost, it is necessary to provide management with a view of the available alternatives and some sense of the risk of the estimating process. A comprehensive evaluation would consider these factors by developing, for each alternative proposed, a structure of operating costs such as that illustrated in Figure 5.

For each alternative (including, as previously stated, the existing system), develop optimistic, most likely, and pessimistic estimates of operating cost for the range of possible business volumes (as defined by lowest possible, anticipated, and highest possible volumes).



*As projected by corporate business plans and budgets.

Figure 5. Estimating Costs for Volume Levels

This approach will illustrate for management the fixed cost base, the maximum costs, and the anticipated costs for each alternative with an insight into the risks of each of these estimates Later, when benefits are matched, the resulting charts will indicate the expected feasibility of the system at various business levels.

It is *not* recommended that all this detail be presented to management. This is working documentation. Estimates may be summarized for presentation to management by:

- Averaging optimistic, most likely, and pessimistic estimates.
- Weighted averaging
- Expressing confidence factors or ranges (in terms of plus and minus percentages) to the most likely estimate or to an average.
- Applying probability theory calculations during the estimating process to arrive at a single estimate carrying an expressed level of probability. This method has been used by some large com-

estimates is significantly better than other methods is not conclusive.

No analysis can be considered complete unless it includes the previous estimates projected over some time into the future. Unless all implementation costs may be recovered within the new system's first year of operation, estimates should cover several years. Again, *the guide is the estimated life of the system or the time necessary for payback of the implementation cost, whichever is less.*

Figure 6 illustrates an operating cost worksheet format that may be used for assembling estimates,[4] while Figure 7 shows a summary of operating cost estimates over a 5-year period.[5]

Project Development and Implementation Costs

A detailed examination of techniques for estimating the costs of application development and implementation is presented in two other portfolios, 34-01-05 and 34-01-06. This section will therefore be a brief summary of the content of the estimate and methods of presentation.

The following major areas of implementation cost must be considered

- Systems design and programming effort through final systems test. Note that it may be desirable to include in this cost the maintenance effort required for the first few months of operation, since this is the "debugging" phase of the project.
- System modifications are treated as a separate item for those instances where the alternative is to purchase a software package.
- Preparation for conversion, which includes review and cleanup of existing files (manual and automated). This may also include planning for organizational change or acquisition of temporary help for the conversion period.
- Development and publication of new and revised policies, job descriptions, and operations manuals for user personnel.
- Conversion of manual files to machine-readable form or conversion and reformatting of existing computerized files.
- Training of the personnel who will utilize the system. Special training of analysts or programmers may be included with design and development cost.
- Pilot operation of the new system or parallel operation of the new system concurrently with the old.
- Miscellaneous project tasks
- Capital expenditures for equipment, software, or specialized forms for conversion, etc.

Figure 8 presents a sample worksheet for a project on which a package was used to implement a new application.[6] These costs may be summarized and spread over the life of the project to provide management with a view of how the project will be budgeted.

Figure 9 shows two ways of summarizing project cost.[7] The summary of total costs includes computer time charges at the in-house rate and approximately half of the project management costs (which would simply be otherwise allocated if this project were not done). This chart is intended to enhance the comparison between alternatives for use of

Elements of Annual Cost	Present System		Alternative System					
			Optimistic		Most Likely		Pessimistic	
	Employees	Amount	Employees	Amount	Employees	Amount	Employees	Amount
Data Processing	14 6	$ 254,600	2 5	$ 30,000	3	$ 36,000	4	$ 48,000
Supplies & Forms		13,300		10,700		11,500		12,800
Operating Personnel	94	583,425	76	482,175	85	527,550	90	561,825
Overhead		252 817		208,942		220,602		243,457
Maintenance Cost								
Personnel	0.2	4,800	0 3	7,200	0 3	7,200	0 3	7,200
Computer Time		17,000		10 000		11,700		12,300
Other								
Amortization of Special Equipment or Software				7,600		7,600		7,600
Subtotal	108 7	$1,125,942	78 8	$756 617	88.3	$830,552	94.3	$893,682
Amortization of Implementation Cost				26,638		23,034		35,229
Final Total	108 7	$1,125,942	78 8	$783,255	88.3	$853,586	94 3	$928,911

Year	Present System		Alternative System					
			Optimistic		Most Likely		Pessimistic	
	Employees	Amount	Employees	Amount	Employees	Amount	Employees	Amount
1	103 7	$1,125,942	78 8	$ 783,255	88 3	$ 851,586	94 3	$ 928,911
2	119 5	1,238,536	81 9	822,418	91.8	904,801	99 9	1,003 224
3	131 4	1,364,389	85 2	863,539	95 5	959,089	105 9	1,083,482
4	144 5	1,499 028	88 6	906,715	99 3	1 016,635	112 3	1,170,160
5	158 9	1,649 491	92 2	1,045 336	103 3	1,077,633	119 1	1,263,773
Total	158 9	$6 873,986	92 2	$4,421,263	103 3	$4,811,744	119.1	$5,449 550
			Capital Expenditure		$88,000			

Figure 7. Operating Cost Summary

the resources available. The second chart is a summary of marginal or out-of-pocket costs which excludes allocated costs and is used to determine the economic feasibility of the project.

The implementation costs appearing in the out-of-pocket cost summary will be used to generate summary figures (by averaging, etc.) for amortizing project cost on the operating cost worksheet, for cash flow analysis, and for payback analysis.

ANALYSIS OF BENEFITS

Benefits are usually classified as tangible and intangible. There is a further "gray area" in which intangible benefits may be given some tangible coloring. These three forms of benefits can be determined and presented as described in the following.

Tangible Benefits

Tangible benefits are derived from two sources — cost reduction in operations and profit improvement by access to new sources of revenue. Cost reductions in operations are readily identifiable during the process of projecting the cost of alternative systems for an application. Profit improvement may result from creating new revenues.

Some examples of these sources of revenue are:

- Creating the capacity for handling increased volumes of business that are readily available
- Creating salable by-products of information generated by the system (name and address listings, property transactions, etc.).
- Creating salable software as a result of developing the system.
- Improving cash management to enable a program of short-term investment (this can be done by improving cash flow, as with a new invoicing system; or conserving cash, as with a new inventory control system that improves inventory turnover).

These kinds of benefits are tangible enough to have been identified and included in the previous operating cost analyses as "earnings."

On this basis, the tangible benefits of proposed alternatives can be determined by comparing the projected costs of the present system with projected costs of implementing and operating the alternative system over a period of time

Together, these figures may be used to construct a cash flow analysis comparing the alternatives as illustrated by Figure 10 (Note that the cost of continued operation of the present system is a part of the cash flow of the alternative until the new system has been implemented) *

As this example indicates, the tangible benefit of this alternative is $1,785,901 over a 5-year period When the benefit amounts are large enough, this type of analysis may be sufficient However, when the benefits determined are smaller in contrast to the investment, so that an adequate return on the investment is in question, or when the pattern of cash flow between alternatives is different but the resulting benefits summarize to near the same amount, a technique called "present value analysis" should be applied. This technique, which uses an internal rate of return to make future cash flows comparable in terms of present dollars, can indicate substantial differences between alternatives

	Cost Elements									
Project Tasks	Systems Analysis	Programming	Keypunch	Computer	Forms & Supplies	Outside Contract Services	User Administration	User Personnel	Project Management	Total
Systems design and programming										
System modifications	$13,500	$16,000		$ 5,200			$ 80			$ 34,780
Preparation for conversion			$ 1,600	1,250		$8,200		$11,860		$ 22,910
Clerical and operating procedures	9,750	750			$1,000			840		12,340
File conversion	2,675	2,050	9,660	16,075			200	3,350		34,010
Training	1,875						240	1,560		3,675
Pilot and parallel operation			900	5,250			540	1,400		8,190
Other project tasks									$14,800	14,800
Subtotal	$27,800	$18,800	$12,160	$27,775	$1,000	$8,200	$1,060	$19,010	$14,800	$130,605
Capital expenditures		38,000								38,000
Final total	$27,800	$56,800	$12,160	$27,775	$1,000	$8,200	$1,060	$19,010	$14,800	$168,605

Figure 8. Implementation Cost Worksheet

Finally, a payback analysis should be made. The payback period is determined by the point at which the cumulative savings (loss) amount becomes a positive figure. Figure 11 is an example.

Intangible Benefits

Truly intangible benefits, which may include such items as improved decision-making, more accurate information, improved legibility of reports, better employee morale, etc. should be recognized and presented to management in a separate section of the cost/benefit analysis. This same section should address other intangibles such as potential risks to the success of the project, the impact of potential project personnel turnover, the management guidance necessary to project personnel to assure success, and other such factors.

Summary of Total Costs						
	First Quarter	Second Quarter	Third Quarter	Fourth Quarter	Fifth Quarter	Total
Optimistic	$39,921	$33,682	$31,346	$36,346	$27,410	$168,705
Most Likely	45 909	38 734	36 044	43 897	30,120	194 704
Pessimistic	52,795	44,544	41,450	52 582	34,050	225,421

Summary of Out-of Pocket Costs						
	First Quarter	Second Quarter	Third Quarter	Fourth Quarter	Fifth Quarter	Total
Optimistic	$31,545	$27,670	$19,865	$32,945	$21,165	$133,190
Most Likely	36 231	31 820	22,844	37,887	24,340	153 172
Pessimistic	41 723	36,593	26,270	43 570	27,991	176 147

Figure 9. Implementation Costs

Borderline Benefits

The term "borderline benefits" is applied here to those benefits that are seemingly intangible, but for which a tangible value with some level of reliability can be determined with sufficient effort. Evaluating these benefits may be as simple as identifying areas of potential cost avoidance or complex enough to require operations research techniques.

One of the better techniques (which has the added convenience of being easy to use) is *Bayesian analysis*. In this approach, if the decision-maker is uncertain about the value of a parameter, he or she should:

- Consider the set of possible values the parameter may take on for the period(s) in question
- Assign a subjective probability weight to each of the possible values of the parameter. (Remember that the total probability for the occurrence of all possible events in a set [universe] is 1.)
- Calculate the expected value of the parameter and use this expectation as the estimate.

Year	Present System	Alternative System			
		Implementation	Operation	Earnings	Total
1	$1,125,942	$164,584	$1,125 942	—	$1,290,526
2	1,128,536	30,120	921,676	$ (6 000)	945,796
3	1,362,369	—	891,998	(28,000)	863,938
4	1,498,628	—	945,517	(33,000)	912,517
5	1,648 491	—	1,002 248	(37 000)	965 248
Total	$6 763 986	$194 704	$4 887 381	$(104,000)	$4,978,085

Figure 10. Cash Flow Analysis

In practice, suppose that we are trying to derive an estimate for the effect on profit that will result from improved information used in decision-making. The user may feel that three values are possible — no improvement in profit having a probability of 0 05, $30,000 improvement with a probability of 0.80, and $50,000 with a probability of 0.15. The estimate would be the result of summing the products of each value multiplied by its assigned probability.

Expected increase in profit $	Probability of occurrence	Expected return $
0	0 05	0
30 000	0 80	24 000
50,000	0 15	7,500
	1 00	31,500 Estimate

If several users are to be consulted on the same estimate, the results from the Bayesian analysis can be averaged (so long as the report makes clear the process used to construct the estimate).

Year	Present System	Alternative System		
		Total Cost	Saving (loss)	Cum. Saving (loss)'
1	$1,125,942	$1,290,526	$(164 584)	$(164,584)
2	$1,238,536	$ 945,796	$ 292,740	$ 128,156

Figure 11. Payback Analysis

Individual estimates can be further refined (as part of a group of estimates) by using the Delphi technique, which consists of an iterative series of responses from members of the group being interviewed. After the first estimates are obtained, they are collated, and the extremes (range) and mean are determined A second round of interviews is conducted in which each member of the group is separately informed of the collated results obtained in the first round and asked for a new estimate. It has been found that three or four such iterations will result in an increasingly congruent set of responses that have a high degree of probability based on the varied knowledge and viewpoints of the individuals participating.

The estimates created by the foregoing methods (when properly noted as to source and content) may be included as "earnings" or

GETTING IT TOGETHER

Presentation of the cost/benefit analysis should be made in a concise format oriented toward decision-making.

The following structure is suggested:

(1) Summary.

The summary should be a one- to three-page section that includes, for all desirable alternatives and the continuation of the present system, a cash flow analysis, and computation of the payback period and return on investment (savings plus earnings divided by implementation cost for each year of the projected period) for the proposed new systems alternatives.

(2) Cost/Benefit Analysis.

This section should contain the supporting detail for:

- Present system operating cost projection.
- Alternative systems operating cost projection.
- Implementation cost for alternatives.
- Earnings projected (tangible and "borderline").
- Savings projected.
- Sources of information.

(3) Intangible Benefits.

The intangible benefits determined for the alternatives should be noted separately for each alternative and the sources of such information indicated. If some of the intangible benefits are key to the proposal, they may be indicated in the summary section and repeated here with supporting detail.

RECOMMENDED COURSE OF ACTION

The cost/benefit analysis is usually the single most important determinant in the project selection and priority setting process. A comprehensive approach to cost/benefit analysis can and should influence design considerations, selection of realistic alternatives, and the final management decision-making process.

While technical feasibility and design of the alternatives are usually the province of the analyst and the DP function, it is in the area of the cost/benefit analysis that company management has the opportunity to exercise its knowledge of corporate goals and objectives and its decision-making powers for the good of the overall organization.

This portfolio was written by Louis Fried consultant.

References

1. Gregory, R H and Van Horn, R L., *Automatic Data Processing Systems: Principles and Procedures*, Wadsworth Publishing, Belmont CA 1963, pp 552-595.
2. This section is derived from J. Emery's "Cost/Benefit Analysis of Information Systems," *The Society for Management Information Systems*, 1971
3. Ibid
4. Fried, L., "How to Analyze Computer Project Costs," *Systems Analysis Techniques*, ed., J.D. Couger and R.W. Knapp, Wiley NY, 1974, p 495.
5. Ibid.
6. Ibid., p. 493
7. Ibid.

INSIDE:
Objectives, Requirements, Use of Preprinted Forms, Alternative Approaches, Elements of a Feasibility Study Proposal

INFORMATION SYSTEMS ANALYSIS

Selecting and Analyzing Alternatives

Conducting the Feasibility Study

PAYOFF IDEA. Conducting a feasibility study and producing the feasibility report are complex tasks that can provide management with information as to whether a potential project will be of real benefit to the organization. A thorough study can prevent a systems manager from recommending development of a system that is impossible to implement or too costly to be justified. This portfolio discusses identification, qualification (i e , recognizing impractical development solutions and discarding them), and quantification (i e , determining scope) of proposed systems development projects

PROBLEMS ADDRESSED

A feasibility study defines the objectives of a development effort, the alternate paths that effort might take, and the costs and benefits the organization should realize if the project is continued.

It is important to the success of any project, but especially to a project that involves considerable expense, many participants, and/or marginal benefits, that the feasibility study be given proper attention; however, there is a tendency among DP users to seek immediate solutions without really defining the problems to be solved. A thorough feasibility study, one that explicitly defines problems and feasible solutions, can save months or years of project time and avert the professional embarrassment of an aborted project that was ill conceived from the start.

Essential to an adequate and thorough feasibility study is the preparation of a feasibility study report. In conducting the feasibility study, the project manager or systems analyst has supposedly gained an understanding of the problems to be solved and of the nature and cost of possible solutions. In addition, some conclusion has been drawn about the feasibility of each of those solutions. That information must be communicated to management, the users, and the other participants in the project

FEASIBILITY STUDY OBJECTIVES

In response to the question, "Why conduct a feasibility study?" one corporate president answered succinctly, "I don't step into a hot bath without testing the water with my toes first!" From a corporate level, this is a fair response, however, from the viewpoint of the systems development manager, the answer is more complex.

Since the results of a feasibility study are usually reported to corporate management for a decision on whether to proceed with a project, the feasibility study should determine whether the project is justified. Although most requests for systems development or redesign are genuine, the systems development manager should probe behind the obvious request for any hidden reasons. Such a probe can unburden an overloaded systems department and assist in establishing implementation priorities. Whatever the request, the systems development manager must be careful to avoid some of the not-so-obvious pitfalls. He will encounter user/managers who feel that every department has a right to use the computer or whose departments have their own operational deficiencies and who want to correct them by some new computer system design. In addition, there are those managers who wish to extend their sphere of influence by impressing management with their "technical currency." These and others may be reasons for new systems, but they are not necessarily *justification* for them. Justification may be demonstrated on the basis of one or more of the following considerations:

- Payback — Return on the investment of development and implementation expenses should be adequate to warrant proceeding with the system as an alternative to other potential investments of a company's money
- External pressure — Some development or modification efforts are imposed by external authority and become a requisite of remaining in the business These demands may come from government regulatory agencies, external auditors, supporting banks, insurance companies, and the like.
- No reasonable alternative to the computer exists — Typically, such projects may involve heavy computational loads — for example, analysis of weather data or statistical analyses of large data samples A second consideration may be response time. While there are a number of ways to perform a task, only a computer solution can provide the results in time to be useful. Some high technology process control applications fit this description, but it can also apply to some clerical operations
- Corporate directive — Realistically, not all computer applications may be "justified" in the objective sense Some are the result of corporate policy changes or the directives of management. It should be noted that in spite of the apparent mandatory nature of these "requests," the feasibility study still can have substantial value Policies have been changed or directives rescinded when the promulgators became aware of the total cost and the other effects of their original decisions

Based on the preceding the primary objectives of a feasibility study are to ensure that the project solves a real problem (as contrasted to a management whim) and to ensure that the problem is worth solving with the use of a computer (i.e., that there is sufficient return on investment). Additional objectives are:

- To provide management with a basis for allocating corporate investment and DP resources among competing demands and thus provide information for setting a priority for the project.
- To ensure that the proposed project is consistent with corporate objectives and goals and with corporate and DP long-range plans (any inconsistencies should be resolved by management).
- To acquaint management with any potential risks of the project. These risks may involve unknown elements in the conceptual design of the system, estimates of the cost and duration of the project, introduction of new software, acquisition of appropriate development personnel, and the capacity of existing or proposed hardware to support the application.
- To provide, when possible, more than one alternative to the current methods used in the application area (in order to give management a wider range of choices).
- To demonstrate the ability of the DP organization to develop and implement the application successfully. This objective may be achieved by having a thorough knowledge of the application area, a good grasp of the hardware and software considerations, the informed participation and consent of the user, a tentative plan for providing the appropriate expertise to the project team, and a high quality of the feasibility study report.

REQUIREMENTS

Before a feasibility study can begin, several management conditions should be met within the organization. Both systems development and user management should agree that a problem needing a computer solution exists and that solving it appears worthwhile. Projects proposed by systems development alone, without user consent and support, rarely succeed.

Since feasibility studies themselves cost money, there should be a company policy supporting feasibility studies for given levels of project work. The usual progression of events leading to a feasibility study is:

(1) Recognition of the problem.

(2) Request for systems development service.

(3) Brief evaluation of the request. This process is generally less than one week of effort. It is designed to screen requests for feasibility and approximate the effort required to meet the request.

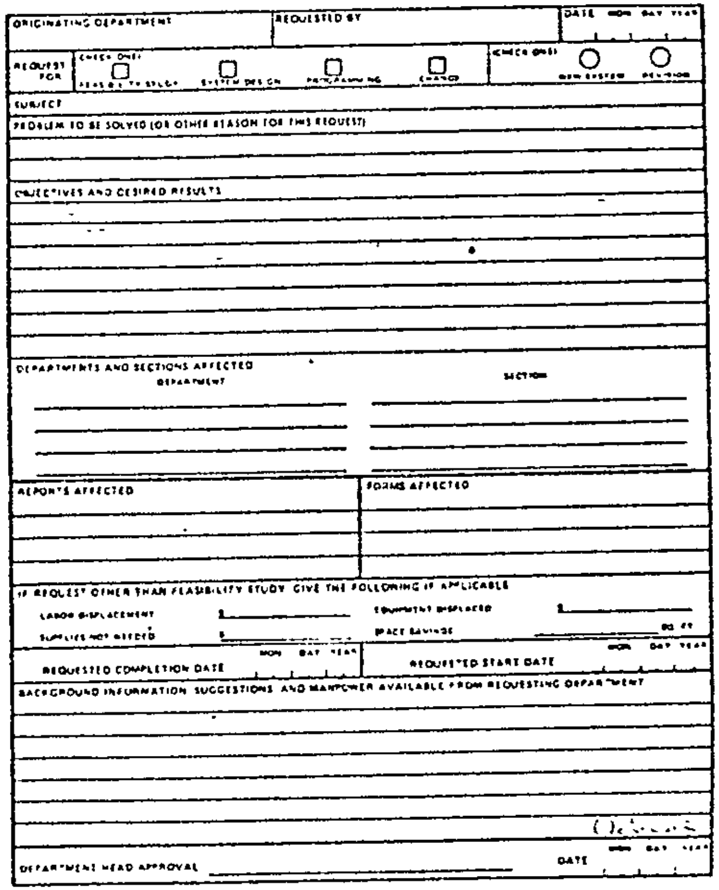
(4) Response to requestor. This short report to the requestor contains four possible answers:

- The request is not practical or is denied for other reasons.
- The request is small, feasible, and will be scheduled if the attached estimate is approved.
- The proposed project is mandatory and a cost estimate and schedule are being developed.
- The request appears to be of a size or complexity that requires a feasibility study. (If this response is applicable, a schedule for providing a proposal for a feasibility study should be established.)

Request for Systems Development

Figure 1 shows a typical request form used for requesting systems development service. The most important aspect of this form is that it forces the requestor to set down in writing the reason for the request and the objectives and results he wants to obtain.

It further requires him to think about the other departments, sections, reports, and forms that may or will be affected by the requested change. In addition, the form requires department-head approval. This ensures that the request has been seen and accepted by the department head and is not just a whim of some lower level employee.



ORIGINATING DEPARTMENT

REQUESTED BY

DATE MON DAY YEAR

REQUEST FOR — CHECK ONE: □ FEASIBILITY STUDY □ SYSTEM DESIGN □ PROGRAMMING □ CHANGE

CHECK ONE: ○ NEW SYSTEM ○ REVISION

SUBJECT

PROBLEM TO BE SOLVED (OR OTHER REASON FOR THIS REQUEST)

OBJECTIVES AND DESIRED RESULTS

DEPARTMENTS AND SECTIONS AFFECTED

DEPARTMENT

SECTION

REPORTS AFFECTED

FORMS AFFECTED

IF REQUEST OTHER THAN FEASIBILITY STUDY GIVE THE FOLLOWING IF APPLICABLE

LABOR DISPLACEMENT $

EQUIPMENT DISPLACED $

SUPPLIES NOT NEEDED $

SPACE SAVINGS SQ. FT.

REQUESTED COMPLETION DATE MON DAY YEAR

REQUESTED START DATE MON DAY YEAR

BACKGROUND INFORMATION SUGGESTIONS AND MANPOWER AVAILABLE FROM REQUESTING DEPARTMENT

DEPARTMENT HEAD APPROVAL

DATE MON DAY YEAR

Figure 1. Feasibility Study, Systems Project Request Form

Request forms will rarely be complete when received, but they usually provide a good start. The form, if incomplete, can be filled in at the first interview with the requestor. If major changes are necessary, a new request should be written and submitted to the department head for approval.

Figure 2 shows a typical form used to evaluate the systems development request, to obtain approval for a feasibility study, and to estimate the costs for systems design, programming, and implementation. This evaluation is initiated by the systems group, the form is used by that group or by the feasibility study team to document concisely, ideally on one sheet of paper, the projected costs of the system — both original or nonrecurring costs and the recurring costs of the system plus the projected savings. Further, and more important, the evaluation form serves as a checklist on the coverage of many items, such as procedures, documentation, user training, and follow-up that are frequently omitted in a feasibility study.

Although preprinted forms are not required for feasibility study requests or evaluations, they help make the job easier and more comprehensive. Forms do not, however, replace a feasibility report. They must be backed up by the detail costs and methods of computation, advantages and disadvantages, intangible values, and so on. They do, however, provide a capsule look at the economics of the proposed project and serve as a sign-off sheet for approval.

Four Basic Approaches

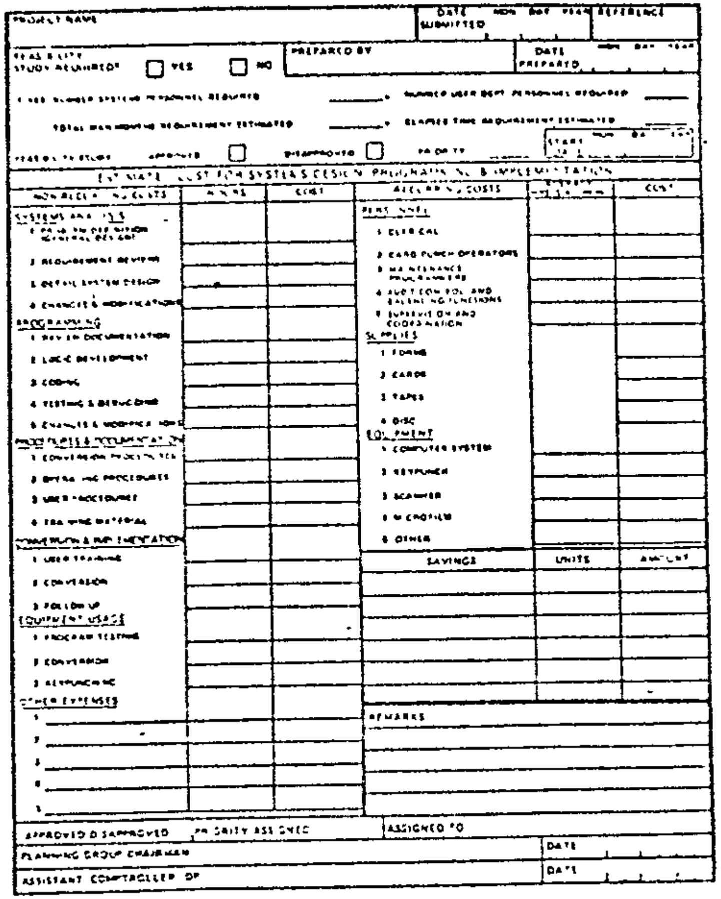
As the systems group considers a feasibility study, there are four basic alternatives to the request for system implementation:

- Develop a computer system.
- Develop a manual system.
- Develop an integrated manual and computer system.
- Do not change the current method.

Develop a Computer System. In these days of faster and less expensive processors, virtual storage, large disk files, multiprogramming, distributed processing, and improved programming techniques, a computer may be the most obvious (although not necessarily the best) solution. Generally the request for a system is proposed on the assumption that a computer system will be designed and implemented.

Develop a Manual System. When a new system must be designed, it is often less expensive and simpler to develop a manual system and operate it at least until it is error free. In addition, performing the job manually allows the user to develop a better understanding of what the system entails. Historical information can be accumulated, and the initial data base can be developed. Later automation of the system, as experience and volume grow, may offer the most efficient solution.

Systems people should not be blinded by the computer. Sometimes modifications to a manual system can accomplish corporate purposes.



PROJECT NAME	DATE SUBMITTED MON DAY YEAR	REFERENCE
FEASIBILITY STUDY REQUIRED? □ YES □ NO	PREPARED BY	DATE PREPARED MON DAY YEAR

EST. NUMBER SYSTEMS PERSONNEL REQUIRED ________ NUMBER USER DEPT. PERSONNEL REQUIRED ________

TOTAL MAN-MONTHS REQUIREMENT ESTIMATED ________ ELAPSED TIME REQUIREMENT ESTIMATED ________

STATUS OF STUDY APPROVED □ DISAPPROVED □ PRIORITY ________ START MON DAY YEAR

ESTIMATE COST FOR SYSTEMS DESIGN, PROGRAMMING & IMPLEMENTATION

NONRECURRING COSTS	HOURS	COST
SYSTEMS ANALYSIS		
1 [illegible]		
2 REQUIREMENT REVIEW		
3 DETAIL SYSTEM DESIGN		
4 CHANGES & MODIFICATIONS		
PROGRAMMING		
1 REVIEW DOCUMENTATION		
2 LOGIC DEVELOPMENT		
3 CODING		
4 TESTING & DEBUGGING		
5 CHANGES & MODIFICATIONS		
PROCEDURES & DOCUMENTATION		
1 CONVERSION PROCEDURES		
2 OPERATING PROCEDURES		
3 USER PROCEDURES		
4 TRAINING MATERIAL		
CONVERSION & IMPLEMENTATION		
1 USER TRAINING		
2 CONVERSION		
3 FOLLOW UP		
EQUIPMENT USAGE		
1 PROGRAM TESTING		
2 CONVERSION		
3 KEYPUNCHING		
OTHER EXPENSES		
1		
2		
3		
4		
5		

RECURRING COSTS	[illegible]	COST
PERSONNEL		
1 CLERICAL		
2 CARD PUNCH OPERATORS		
3 MAINTENANCE PROGRAMMERS		
4 AUDIT CONTROL AND BALANCING FUNCTIONS		
5 SUPERVISION AND COORDINATION		
SUPPLIES		
1 FORMS		
2 CARDS		
3 TAPES		
4 DISC		
EQUIPMENT		
1 COMPUTER SYSTEM		
2 KEYPUNCH		
3 SCANNER		
4 MICROFILM		
5 OTHER		

SAVINGS	UNITS	AMOUNT

REMARKS

APPROVED DISAPPROVED	PRIORITY ASSIGNED	ASSIGNED TO
PLANNING GROUP CHAIRMAN		DATE
ASSISTANT COMPTROLLER OF		DATE

Figure 2. Féasibility Study: Systems Project Request Evaluation Form

Develop an Integrated Manual and Computer System as a Solution. With any computer system, there is some manual effort in its initial processes, but it may be determined that other intermediate steps or validation techniques could be performed more economically.

Occasionally when computerizing a system one small part may require an inordinate amount of time for programming and testing or an inordinate amount of hardware resources. Manual implementation may then be a better solution.

For example, a computer system that processes over 500,000 items a day may have 137 with peculiar characteristics. These 137 items might require three weeks of extra systems design, one and a half months of programming, 20,000 bytes of core memory, and numerous machine cycles. The payback period of automating this part of the job would be more than 100 years, which is, in most cases, unjustifiable.

Do Not Change the Current Method. Frequently a manual system or a manual/computer system cannot be improved enough to justify the expense involved in the change A system should not be changed simply for the sake of change.

If the systems development manager is to fulfill his responsibility, it is important that he select or recommend the method that is most beneficial to the company, whether it be computer, manual, or a combination thereof. This cannot be done by viewing only cost savings as the determiner; other factors must also be considered.

ELEMENTS OF A FEASIBILITY STUDY PROPOSAL

A feasibility study is a project in itself. The proposal for a feasibility study should contain the following elements:

- Objectives — A list and brief explanation of the objectives of the feasibility study, the subjects to be covered, major questions to be answered, and why they are important to the design of the system.
- Methods — A description of how the major questions will be answered in terms of the procedures to be used, observations, and units of measure (if the data is objective and quantifiable). If the data is subjective and qualitative, an indication of who will be interviewed (level, title, function) and what processes will be observed should be included.
- Schedule — Where and when the survey will take place.
- Personnel requirements — The number of people required for the feasibility study, their work classifications or titles, and the area from which they will be provided (systems development user department, consultant, other)
- Skills and training — For each of the above personnel, the skills or knowledge required and/or any training necessary to make up deficiencies If possible, a training schedule should show any outside costs and who would do the training
- Estimated time and cost — Schedules and cost estimates covering direct labor, travel expenses, any data reduction (processing and analysis), and report preparation.
- User participation — The user's role in consulting with and familiarizing systems development analysts with the business operation Offices, files, and information to which the study team must have access should be listed Any other support that will be expected of the user and a tentative schedule of user personnel time requirements should be provided. If the user will be involved in special data-gathering efforts, a sample of the reporting format(s) should be included.

After the user approves the feasibility study, it should go to top management or a steering committee for approval before work can begin.

As with any other project, consideration must be given to project team organization, project planning, and control. Concluding the study within estimated time and cost helps create credibility in later project development work

CONDUCTING THE STUDY

A large part of conducting the feasibility study depends on the gathering of information. This information is crucial to the activities involved in conducting the study. In addition, the information is useful for later documentation and preparation of the feasibility report. Although personal observation of current methods and general research can provide some of this information, more in-depth information can be obtained by such methods as round-table discussions (including brain-storming sessions among members of the study team), interviews with information sources in the company, discussions with equipment manufacturers, consultations with outside specialists, and discussions with software firms. Also of value is information provided by the user department and its study team representatives.

The activities and questions involved in conducting a feasibility study are:

(1) Training and orientation of project personnel, including setting up the project team and project control mechanisms.

(2) Refining the objectives of the study by projecting the objectives of the proposed system; spelling out exactly what is to be accomplished by solving the problem or completing the project. Some objectives of a new system might include:
- A faster process (e g., earlier reports, bills, and refunds)
- A more effective operation (e.g., giving supervisors more time for supervising by requiring less of their time to correct defects)
- A better end product (e.g., the same quality at less cost, better quality at the same cost, or better quality at less cost)

(3) Collecting present system documentation, if any.

(4) Observing and documenting present system flow, job steps, decision elements, inputs and outputs.

(5) Identifying the cost of the present operation, the effects of business volume on costs, the effects of inflation on costs, and so on.

(6) Projecting the costs of continued operation of the present system, including any costs of adverse impact on the business from its continued operation. Present and projected costs should include the following:
- Operating personnel
- Overhead
- Computer costs (if any)
- Maintenance costs
- Supplies
- Amortization
- Business costs

(7) Determining the information requirements by establishing the requirements for processing, decision making, and reporting.

(8) Determining any constraints on the system design (such as legal requirements, response time requirements, data inquiry needs, etc.).

(9) Defining file content requirements (data elements, tentative file organization and access methods, etc.)

(10) Developing alternative system concepts and providing a narrative synopsis and flowchart of each.

(11) Identifying and defining preliminary hardware and software requirements for each alternative

(12) Estimating the costs and possible schedules for each alternative The following major areas of implementation cost must be considered

- Systems design and programming effort through the final systems test Note that it may be desirable to include in this cost the maintenance effort required for the first few months of operation, since this is the debugging phase of the project
- System modifications, if the alternative is to purchase a software package.
- Preparation for conversion, which includes review and cleanup of existing manual or automated files. This may also include planning for organizational changes or acquiring temporary help for the conversion period.
- Developing and publishing new and revised policies, job descriptions, and operations manuals for user personnel.
- Converting manual files to machine-readable form, or converting and reformatting existing computerized files.
- Training the personnel who will use the system. Special training for analysts or programmers may be included with design and development cost.
- Pilot operation of the new system or parallel operation of the old and new systems.
- Miscellaneous project tasks.
- Capital expenditures for equipment, software, or specialized forms for conversion, etc

(13) Estimating the operating costs of the proposed alternatives, including all elements in item 6. In addition, considering any business benefits that will result from implementing the new system

(14) Documenting the intangible benefits of each alternative and the risks of each approach.

(15) Preparing the feasibility report

(16) Reviewing the report with the user and making any necessary revisions.

(17) Presenting the feasibility report for management approval.

DOCUMENTING THE FEASIBILITY STUDY

All the data gathered in the feasibility study would be overwhelming if presented in the final report, however, it does have considerable value if the proposed project is approved All project documentation should, therefore, be well organized and filed at the conclusion of the study.

Systems analysts are usually responsible for documenting the feasibility study. The report should contain all the facts relating to the process of examining the systems selection There is no absolute number of pages in the report, rather, the magnitude of the project dictates the size of the report

A number of factors should be kept in mind when considering the report style. Report compilers should strive for brevity and economy. With this in mind, all inclusions should be fully treated and care should be exercised to avoid items that are not properly defined. Although this is a technical report, unusual technical language and abbreviations should be avoided, since the report will be circulated to people with varying degrees of technical competency. The information on the report will come from many parts of the firm and consequently the responsibility for assembling the documents, recording the activities, and formalizing the final report should be assigned to a person who has broad knowledge in related areas

Report preparation can be greatly improved by having the rough draft edited If an editor or technical writer is not available, a member of the development staff or a manager with good writing skills can serve this purpose. Many factual, accurate, and thorough reports have been given short shrift because of poor grammar, inconsistent writing style, and/or the excessive, confusing, or ambiguous use of personal pronouns. In addition, an editor is usually aware of the proper perspective from which the document should be written and thus changes first-person singular and future-tense presentation.

The Report. The feasibility report itself should consist of three major sections:

- A brief management summary of the conclusions of the study. This section should be as short as possible yet should convey the major objectives of the proposed system, the conclusions, and the key reasons for reaching those conclusions. Ideally this section is one to five pages long and preceded by an approval page for appropriate signatures
- The body of the report
- Any appendixes to the report.

The body of the report contains the supporting documentation for the management summary. It, too, should be as concise as possible, but its length is a function of the scope and complexity of the system, and the number of alternatives examined A consistent format should be used throughout for easier reading Alternatives should be presented in sections identically organized so that they can be compared readily Charts or graphs describing similar data or events should use the same formats for ease of comparison.

The body of the report should contain the sections described below.

- Introduction — This section briefly outlines the genesis of the study, identifies the requestor(s), and delineates the initial objectives.
- Problem statement — This section briefly outlines the problem that the proposed system is to solve
- Constraints — This section describes any constraints on the company or on systems development that influence the system design alternatives. These constraints may be legal, technical, corporate policy, or business (e g , limitations on capital investment, limits on manpower budgets, tax considerations, present hardware or software capabilities).

- Scope of the system — This section specifies the functional areas the system addresses, whether this is a new application or a modification, and which departments will be affected and in what manner. In addition, any changes in corporate policy or procedures necessitated by the implementation of the system should be included, as should the approvals required to proceed with the project.
- Desired results — In general, this section describes the results desired by the user. It includes such considerations as response time, service levels, size of user staff, net change in user operating costs, increased information visibility, and so on. It should be noted that this section describes the user desires only, not the final results of each alternative, as these may represent some compromise with desired results.
- Alternatives — The following sections should be repeated for each alternative presented. They should be arranged so that each alternative is presented as a complete picture, from concept to tentative schedule.
  - Solution concept. This is a brief (one- to three-page) description of the major features of this alternative solution. It identifies both manual and automated processes and the general flow of the system If necessary, a top-level flowchart may accompany this section.
  - Cost/benefit analysis. For a complete view of the contents of this section, refer to portfolio 32-03-04, "Performing Cost/Benefit Analysis." This section contains such supporting detail as implementation cost of the alternative, conversion costs, operating cost projections, any earnings projected, savings projected, and intangible benefits.
  - Hardware and software impact This section describes the effect in the present configuration of hardware and software. For example, any additional equipment or required additions or modifications to systems should be listed with an explanation as to how they would be provided.
  - Conformity with corporate plans. This section indicates whether the proposed alternative conforms to corporate goals and long-range plans If the alternative does not conform, short- and long-term effects of this alternative should be listed.
  - Schedules and staffing. A tentative implementation schedule for the alternative and how the project team would be staffed should be organized using a bar chart or similar illustration.
  - Risk. Any risks involved in this alternative, including those pertaining to schedule estimated cost, systems software modification, and hardware reliability or limitations, should be described
  - User participation This section describes involved user personnel and the extent to which they must contribute to ensure the project's success
- Summary and recommendation — This section summarizes for all alternatives presented and for the continuation of the present system: a cash flow analysis, computation of the payback period, intangible benefits, and risks. A concluding paragraph should indicate the project manager's recommendation as to the appropriate course of action.

- Appendix — This section should contain any technical or financial detail that supports the feasibility study report.

RECOMMENDED COURSE OF ACTION

A feasibility study is a vital part of the systems life cycle. Its primary purpose is to eliminate unwanted surprises for company management. In this context, it may be viewed as an insurance policy; however, it must be used with discretion and in such a manner that the premiums (the cost of the feasibility study) do not exceed the potential loss

The feasibility study should be a careful analysis and projection of the future effort. When conducting a feasibility study, the background information must be fully described, advantages and disadvantages of the proposed alternatives should be given, and the audit and control functions should be considered in conjunction with each alternative. This information, when documented and presented in a well-organized and well-written feasibility report, helps direct the project toward successful completion.

This portfolio was written by the AUERBACH Editorial staff. It is partially drawn from original material submitted by Louis Fried and Jonathan Palmer.

"Hay otros mundos pero están en éste"

Eluard.

CAPITULO I

Los Enfoques de Planeación.

1. Las diferentes concepciones de la Planeación se pueden resumir en las siguientes líneas: (1) *La Planeación* como un proceso de toma de decisiones, (2) *La Planeación* como un Sistema Conceptual, (3) *La Planeación* como una actividad de intervención en la realidad social, (4) *La Planeación* como un instrumento de la Política Económica, (5) *La Planeación* como un corponente de la *Comunicación Social*, consecuentemente, como un componente de un proyecto político.

Cada concepción citada de la Planeación se deriva (i) de una orientación ideológica específica y (ii) de una visuilización parcial de la realidad: de una dimensión particular Factores y razones histórico-político-culturales subyacen en las concepciones referidas. Como no existe, aún, una teoría acabada de la Planeación se observa un alto grado de confusión; en muchos casos es atribuible a la terminología; en otros se debe a la impresión que en cada actividad profesional dejan "*sus propios textos*". "No es improbable hallar equivalencias entre (a) Planeación y Presupuestación; (b) Planeación y Programación; (c) Planeación y Administración; (d) Planeación y Pronóstico; e) Planeación y Proyecto

Económico. Las tres primeras equivalencias se encuentran mas difundidas entre los especialistas de las áreas administrativas; el inciso (d) en las matemáticas aplicadas; la última corresponde a algunas escuelas de economía. En cada caso la equivalencia es errónea: la correspondencia eliminaría muchos aspectos importantes que contiene la Planeación. Tal y como se discute posteriormente.

Paralelamente, la Planeación resulta ser, también, un concepto "comodín" o de usos múltiples para fines diversos. Dentro de la vida cotidiana se escuchan comentarios sobre fallas o deficiencias en ciertas actividades debido; (1) a la omisión de tareas de planeación, (2) a la planeación incompleta o ineficiente, (3) a la falta de articulación entre un plan y la operación del mismo, etc. En algunos casos se desprenden dichos comentarios de manera superficial: búsqueda de un o varios factores a los que se les desea involucrar por ser candidatos deseables para ser "culpables". En otros casos, se derivan de un deseo por *racionalizar* la actividad humana. Por ejemplo, por *controlar* mayormente un determinado proceso o actividad. También se presenta como una actitud defensiva y/o pasiva ante actos de otras personas, instituciones, organizaciones o tomadores de decisión. Los elementos citados son sólo una muestra pequeña de los factores que trazan una explicación pero revelan una característica crucial: *La Planeación incide en el Sistema Social y afecta a diversos agentes de la vida social*. Consecuentemente, contiene siempre un elemento político. La importancia de este punto se describe y ratifica en el contenido de los dos primeros capítulos de esta tesis.

2. La Planeación como proceso de toma de decisiones.

La extensión de la Teoría de Decisiones al campo de la Planeación se manifiesta por una definición de Ackoff: "es una toma de decisiones anticipada".[1] Aunque el propio Ackoff ha propuesto definiciones más completas la acepción anterior ha atraído, quizá por su simplicidad, a un número muy vasto de personas interesadas en el tema. Se debe indicar aquí que toda toma de decisiones es anticipada; *es previa a la acción, actitud, disposición, que se seleccione.* Por lo tanto, el punto crucial es el que se refiere al periodo de anticipación. La discusión aquí, sin más elementos de información sobre el objeto y el sujeto de la Planeación, se presenta como de índole casuística. Hay necesidad de añadir otros aspectos para que la reflexión sea más fructífera. Por ejemplo, el tiempo en que se tarde tomar una decisión; ie. escoger un curso de acción; el tiempo de preparación para llevarla a cabo, el tiempo requerido para evaluar los resultados correspondientes, etc. En suma, tal y como lo señala White,[2] la propuesta es ambigua y/o relativa al caso específico que nos ocupe. Quizá Ackoff no fue tan explícito debido al entendimiento tácito de que el término "anticipada" es relativo al contexto donde se aplique.

Otro punto que llama la atención reside en que la toma de decisión pueda ser singular o única. La Planeación, corrigió después el propio Ackoff, "es un <u>sistema</u> de decisiones interdependientes entre sí que se dirigen a coproducir un futuro deseable"[3] En este enfoque se hace así evidente el carácter dual de la Planeación: <u>como sistema</u> y como <u>proceso</u>.

De hecho, la Planeación se realiza en una imagen de la realidad (sistema conceptual)* en donde se toman decisiones en forma de proceso. Dichas decisiones se traducen en acciones que inciden en el objeto de la Planeación, el que ha sido modelado como sistema. Como es obvio esperar no necesariamente coinciden la decisión y la acción concreta que se lleva a cabo. Una de las razones por la que no se da la coincidencia es la naturaleza distinta del espacio de referencia. En otros términos, el lenguaje y su naturaleza es distinto én cada caso. Afectan elementos denotativos y connotativos que se expresan, p. ej., como variables linguísticas.[5]

En el sistema conceptual, el proceso de decisión es caracterizado por $\{x(t)\}_\tau$, en la intervención en la realidad se obtiene $\{y(t)\}_\tau$ donde

* que algunos autores denominan "<u>constructo</u>". Ver por ejemplo a Gelman.

$\{X(t)\}_\tau \xrightarrow{\Delta} \{y(t)\}_\tau$; Δ actúa como un operador que singulariza la acción correspondiente[6]. Como se puede suponer X(t) indica la acción seleccionada por el decisor (o tomador de decisiones) en el tiempo τ. Si el proceso de decisión tiene un espacio parametral finito o numerable se puede modelar por $\{X_n(t)\}_n$. Aquí hay que notar que los periodos sucesivos de la decisión no tienen porque ser iguales. Los valores de $X_n(t)$ pueden conceptualizarse de diversas formas:

(a) Como variables aleatorias - en el espacio de resultados. Consecuentemente dan lugar a un proceso estocástico;

(b) Como variables deterministas. Esto es como funciones $X_n : f : A \rightarrow A$. Del conjunto de cursos de acción en sí mismo.

(c) Como variables deterministas sumergidas en un conjunto A que puede cambiar de etapa a etapa. En otros términos, A sólo está delimitado en el corto plazo pudiendo modificarse - por ejemplo, por la tecnología o por la amplitud del espacio de acción en cada etapa.

2.1 El primer caso (a) corresponde a la observación de un agente externo al proceso de decisión. Tal es el caso de un observador (investigador) que puede tener como elemento de información: (i) el conjunto de cursos de acción A de indecisor que se considera invariante en cada etapa; (ii) se conocen -o si no se conocen con irrelevantes- las etapas en donde se selecciona un curso de acción. Un ejemplo sencillo es el caso que $A = \{a,b,c\}$ en cada etapa la selección tiene que ser cualquiera de los tres cursos de acción comprendidos en A. La regla de decisión[6] en cada caso (aunque no sea explícita) permite evaluar, etapa por etapa y globalmente, cuál es la "mejor acción". Si el observador citado no tiene otros elementos de información adicionales podría proponer -y comprobar- diversos criterios de decisión que le permitan identificar cuál es el proceso subyacente del decisor al que está investigando.

En este enfoque existe un paradigma[7] implícito: "todo proceso de decisión tiene una determinada racionalidad y esta es descubrible". En la práctica es común encontrar que el investigador hace uso de un paradigma (o una visión del mundo") distinto para explicar -y criticar- el proceso de decisión examinado. Actualmente se observan muchos ejemplos de lo citado:

(i) examinar desde el enfoque marxista, desde la teoría de la dependencia o desde el modelo neoclásico, el gobierno del Presidente López Portillo en el campo económico,

(ii) examinar según las escuelas funcionalista, estructuralista o marxista la política cultural de un gobierno específico;

(iii) examinar desde diferentes perspectivas las opciones de comercio exterior.

El tipo de observación es distinto si se aplica en forma *retrospectiva* (EX POST) que si se aplica en forma *prospectiva* (EX ANTE). Los ejemplos (i) a (iii) coinciden en la evaluación EX POST. Una ilustración del tipo EX-ANTE correspondería a la predicción del proceso de decisión de un decisor ajeno al investigador.

Por otra parte, la forma citada sugiere que el investigador tiene -o prefiere tener- una disposición pasiva o inactiva[8] ante el proceso de decisión que observa. Este caso no será estudiado pues no corresponde al enfoque de la tesis. Es cierto que es un enfoque desde la perspectiva del espectador o investigador académico, por eso mismo tendría otro título y correspondería a otro lugar su discusión.

2.2 El caso (c) es el más apegado a la realidad. En varias ocasiones, básicamente debido a. (i) innovaciones tecnológicas, (ii) discontinuidades abruptas en el campo político o económico*, (iii) *ganancia de información que conduce a enriquecer* A. En cada etapa de decisión es claro que se requiere conocer el contenido de A pero, en etapas previas no necesariamente es así. En otros términos $X_n(t) \Rightarrow A_n$ conocido pero $A_m \neq A_n$ para $m > n$. De hecho $A_n \subset A_m$ $(m > n)$ aunque el valor o la utilidad de cada curso de acción cambie. Es-

* lo que señala Trist como "turbulencias"[9].

te enfoque tiene aún más características de las señaladas·

(a) permite combinar la dualidad "evaluativa-de desarrollo";[10] Así se puedan desarrollar los métodos de Kaufmann[11], algunas técnicas cualitativas ad hoc,[12] etc.

(b) permite analizar el beneficio/costo esperado de la investigación científico - tecnológica.

(c) Cualquier problema de decisión, conforme se amplía su horizonte de planeación, se convierte en un problema no estructurado [13] Con lo cual, los defensores del principio de Ignorancia Completa confirmarían la validez de su supuesto.

(d) posibilita la identificación de factores activos en la Planeación, al establecer que A depende de la intervención de distintos niveles de acción. Se consolida el impacto de la voluntad humana.

2.3. La conceptualización más común, hasta ahora, es la representada por el caso (b) Se presupone aquí que existe un orden de preferencias entre los cursos de acción que obedece a un criterio de decisión. Como se puede directamente observar existen dos grandes caminos en este caso: (1) acudir a la teoría del Valor o de la Utilidad para evaluar numéricamente todo curso de acción y en cada etapa; (2) restringir la aplicación para diserminar cada curso de acción en términos ordinales (con una preferencia o valor implícito). A (1) se le denomina en Decisiones, teoría "fuerte" y a (2) teoría "débil" debido a las exigencias que contiene cada una.

Este es el caso más representativo de la extensión de la teoria de decisiones más tradicional. Como se puede observar la correspondencia entre esta conceptualización y la modelación matemática (vía programación dinámica, teoría de redes o programación multi-objetivos o multi-criterios) es robusta.

2.4 A esta altura de la discusión es importante hacer notar que existe un ingrediente que afecta relevantemente la "*textura*" de este enfoque. En un proceso de decisiones interrelacionadas entre sí. en cada etapa la selección de un curso de acción específico está afectado por las selecciones previas y por sus consecuencias. Los Sistemas Sociales,[14] donde se realiza la Planeación, son Sistemas Abiertos. Por lo tanto, influye el Medio Ambiente o Entorno del Sistema. De aquí que al seleccionar un curso de acción no es posible determinar con certeza a que resultado se va a llegar. Entonces, la decisión se da, por efecto de la relación entre cursos de acción y resultados, en un marco de riesgo o de incertidumbre.

Así se concede[15 que en el corto plazo se pueda planear bajo condiciones de riesgo y, en el largo plazo sólo bajo incertidumbre.

¿Cómo es, entonces, que la Planeación pueda generar elementos de acción social?. ¿Es sólo un espejismo? La defensa de la utilidad, en ese sentido, de la Planeación radica en el enfoque proactivo y en la voluntad de intervenir en la realidad social hasta alcanzar un resultado desecho. Como es claro advertir, así, el ejercicio de la voluntad individual, de un grupo, red o clase social enfrenta voluntades que le son favorables, (alianzas), opuestas (coaliciones) y antagónicas (conflictos). Se está, entonces, en el espacio del poder, en la faceta política indispensable e inherente a la planeación como tal.

De esa forma aparecen varias escuelas de planeación según la modalidad o papel que le asignan al aspecto político:

(1) El aspecto político sólo está presente en los objetivos (Fines, Metas, Ideales) que se seleccionan y persigan.[16]

(2) El aspecto político se configura por las relaciones de poder entre decisores, beneficiarios* y planificadores. Objetivos, cursos de acción y actividades de operación son, exclusivamente, consecuencia de los resultados de las relaciones de poder.[17]

(3) Lo político está presente en todos los aspectos consustanciales al proceso de planeación. Aquí hay tres vertientes generales:

* Incluyen aquí los beneficiarios afectadores.[19]

(a) La política domina, como eje, las actividades de la planeación; no sólo como elemento básico contenido en cualquier "plan de acciones" sino como transformación de estado (búsqueda de propósitos): aspiración de cambios o defensa del *status quo*. Fundamentalmente se revela el contenido ideológico: la manipulación de los valores en la acción humana;

(b) La planeación es un instrumento de la política, más explícitamente, la planeación es un componente de la participación y ésta de la dominación (que, a su vez, lo es de la hegemonía);

(c) La planeación sólo tiene sentido por la acción a la que conduce. Cualquier acción se inscribe en un marco de racionalidad en un contexto de conflicto: el que corresponde al enfrentamiento de clases sociales antagónicas: Para ciertas escuelas sociológicas se darán interrelaciones entre clases dominantes y subalternas de países hegemónicos y/o dependientes (como Touraine). En el enfoque Marxista el conflicto se dirime, como es sabido, en la lucha por la propiedad de los medios de producción.

De esa manera, se pueden considerar -y clasificar- los enfoques de planeación por niveles según los autores principales que los sustentan y definen:

(i) El primer nivel que corresponde a la discusión sobre objetivos;

(ii) El segundo nivel corresponde a la especificación y determinación de las relaciones de poder;

(iii) El tercer nivel tiene tres vértices los correspondientes a los incisos (a), (b) y (c).

Se representan, a continuación, algunos enfoques conocidos que se enmarcan en el planteamiento mencionado.

CUADRO 1

Primer Nivel ------------------------------	Ackoff
Segundo Nivel -----------------------------	Trist
Tercer Nivel ------------------------------	Ozbekhan
Caso (a) --------------------	Aldo Solari
Caso (b) --------------------	
Caso (c) --------------------	Touraine; neo-marxistas

La revisión de lo que es la Planeación -de su contenido- puede realizarse bajo muchos enfoques. Una posible opción corresponde a la visualización de los aspectos normativo, estratégico y operativo. El aspecto *normativo* se refiere a la estructura y reglas lógicas (formas de construcción de proposiciones) sobre la Planeación;

El aspecto *estratégico* se relaciona con lo que la Planeación pretende: hacia donde se dirige (fines, metas, objetivos), cuándo se espera conseguirlo.

Finalmente, el aspecto *operativo* corresponde a la forma como se seleccionan, implantan y controlan las acciones consideradas.

Como se esperaría, existen trabajos específicos que se enfocan a desentrañar los problemas de cada aspecto. Así, por ejemplo, se ha estudiado -en cuanto al aspecto estratégico- (i) cómo derivar objetivos; (ii) como evaluarlos; (iii) como establecer redes de objetivos. Similarmente ha sido importante la contribución en el aspecto operativo: (i) casos y causas de las deficiencias en llevar a cabo planes, programas o proyectos; (ii) elementos de regulación y control de las actividades; (iii) esquemas de correspondencia entre los agentes de planeación que participan en los aspectos estratégico y operativo, entre otros.

Cuando se han referido preguntas al porqué y al para qué de la planeación existe una ambigüedad en la inserción apropiado. Parecería ser, entonces, que el para qué de la Planeación está vinculado al aspecto *estratégico*. No obstante, el *por qué* no tiene ubicación precisa.

Aquí se presentan dos preguntas relevantes: (a) ¿porqué planear?. (b) ¿por qué se planea de una determinada manera o según un enfoque determinado? El primer inciso (a) tiene adheridas las siguientes interrogantes: ¿para qué planear? ¿para quién planear?.

Todo este grupo de preguntas y otras relacionadas parecen ser, en mi opinión, de importancia sustantiva. Sin embargo se dejan de lado o se omiten casi siempre. Hay razones que explican esta circunstancia. El lector puede reflexionar, desde su concepción personal, acerca de varias de ellas. En particular, los aspectos, políticos e ideológicos justificarán o rebatirán su pretendida omisión. Se afirma que dicha omisión es pretendida y no es objetiva puesto que explícita o implícitamente , dichas preguntas están presentes. En este trabajo, y en otros anteriores, se considera su respuesta como *indispensable*. Básicamente porque cualquier proceso, sistema y/o actividad de *Planeación* implica un compromiso. Compromiso de tipo social que está íntimamente ligado con los resultados o impresiones sociales de la Planeación.

¿Quién planea?: Planea quién tiene *la capacidad* de hacerlo. Planea quién tiene *el poder* para hacerlo. De aquí que se requiera el saber y el poder. En este sentido, tanto el *saber* como el *poder* se dan *formal* o *informalmente*. El grupo hegemónico* es propietario del saber formal y ejerce el poder formal. Los grupos subalternos o dominados, si acaso, son capaces de planear en espacios sociales más reducidos. Paralelamente pueden hacer uso de elementos formales si son auxiliados por planificadores comprometidos.

* en el sentido de Gramsci.

CUADRO II(*)

	injerencia en la planeación	posición (en general)	alternativa paradigmática consistente al grupo
Grupo hegemónico	muy alta	Control social (conservación del status quo)	*Funcionalismo: "Humanismo".*
Grupos dominantes	mediana (variable)	Cambio parcial	*Variable*
Grupos subalternos	muy bajo	Cambio (radical)	*Marxismo; Anarquismo Enfoques "Antiorganizativos".*

En general, se hace evidente que el cuadro anterior implica una consideración de racionalidad que pudiese no estar presente. Los paradigmas racionales asociados a los grupos hegemónico y dominantes(**) darán realce a la conciliación. El cambio, para los grupos subalternos interesa revelar en toda su dimensión, al conflicto. No por esto se quiere mencionar que todo grupo subalterno proponga -y defienda- corrientes de conflicto social. El grado de asimilación de ciertos grupos subalternos mediante, p. ej.; la participación formal o la informal es posible que, dadas otras condiciones, sea muy alto. La discusión de este punto es compleja y amplia; dicha amplitud requeriría, al menos, la extensión total de esta tesis.

* esta clasificación, aún no completa, se derivó de una discusión con Raúl Garduño. Subyace aquí una correspondencia con las teorías sociológicas principales. Se reconoce, entonces, la inclusión de la Planeación como "elemento" de la teoría social. La clasificación de grupos extiende la descrita por Gramsci. La clasificación de la "posición" se refiere a los países Capitalistas dominantes o dependientes. Es claro [illegible] los factores que afectan la descripción de la tabla.

REFERENCIAS.

(1) Ackoff, R.L. Un Concepto de Planeación de Empresas
(traducción).
Limusa, México 1972.

(2) White, D.J. Teoría de la Decisión (traducción)
Alianza Editorial.

(3) Ackoff, R.L. Redesigning the Future.Wiley.
Nueva York, 1974.

(4) Ibid.

(5) Zeleny, R. "Some approaches to decision theory"
Human Systems Management
Vol. O. Oct. 1980 Pennsylvania.

(6) Ackoff, R.L. The Art of Problem Solving Wiley.
Nueva York, 1978.

(7) Román, M.F. "Las Orientaciones Políticas de la Planeación Prospectiva". XIII Congreso Interamericano de Planificación SIAP.
Caracas, Venezuela (mineo).

(8) Ackoff, R.L. (1974) op. cit.

(9) Sachs, W.M. "Un Diseño del futuro para el futuro".
Fundación Javier Barros Sierra.
México, 1978.

(10) Ackoff, R.L. et al.Scientific Method.Wiley.
Nueva York, 1962.

(11) Kaufmann, A; Fustier, M. y Drevet A. La Invéntica (traducción)
Deusto. Bilbao, 1973.

(12) Ibid. op. cit.

(13) Ackoff, R.L. (1978) op. cit.

(14) Emery, F. y Trist, E. Systems Thinking. Penguin Books.
Londres, 1966.

(15) Ibid. op. cit.

(16) Ackoff, R.L. (1972) op. cit.

(17) Román, M.F. (1980) op. cit.

(18) Román, M.F. Modelos matemáticos aplicados a la Educación.
Tesis Actuario
Facultad de Ciencias, UNAM. 1975.

(19) Román, M.F. La Planeación y el Derecho.
Conferencia.
UAM-Azcapozalco
Febrero, 1982. (mimeo).

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

DEFINICION DE REQUERIMIENTOS DEL USUARIO
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

V. DEFINICION DE REQUERIMIENTOS DEL USUARIO (DOCUMENTACION COMPLEMENTARIA)

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

TEICHROEW & HERSHEY 23

PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems

I. Introduction

Organizations now depend on computer-based information processing systems for many of the tasks involving data (recording, storing, retrieving, processing, etc.). Such systems are man-made, the process consists of a number of activities: perceiving a need for a system. determining what it should do for the organization, designing it, constructing and assembling the components, and finally testing the system prior to installing it. The process requires a great deal of effort, usually over a considerable period of time.

Throughout the life of a system it exists in several different "forms." Initially, the system exists as a concept or a proposal at a very high level of abstraction. At the point where it becomes operational it exists as a collection of rules and executable object programs in a particular computing environment. This environment consists of hardware and hard software such as the operating system, plus other components such as procedures which are carried out manually. In between the system exists in various intermediary forms.

The process by which the initial concept evolves into an operational system consists of a number of activities each of which makes the concept more concrete. Each activity takes the results of some of the previous activities and produces new results so that the progression

3

eventually results in an operational system. Most of the activities are data processing activities, in that they use data and information to produce other data and information. Each activity can be regarded as receiving specifications or requirements from preceding activities and producing data which are regarded as specifications or requirements by one or more succeeding activities.

Since many individuals may be involved in the system development process over considerable periods of time and these or other individuals have to maintain the system once it is operating, it is necessary to record descriptions of the system as it evolves. This is usually referred to as "documentation."

In practice, the emphasis in documentation is on describing the system in the final form so that it can be maintained. Ideally, however, each activity should be documented so that the results it produces become the specification for succeeding activities. This does not happen in practice because the communications from one activity to succeeding activities is accomplished either by having the same person carrying out the activities, by oral communication among individuals in a project, or by notes which are discarded after their initial use.

This results in projects which proceed without any real possibility for management review and control. The systems are not ready when promised, do not perform the function the users expected, and cost more than budgeted.

Most organizations, therefore, mandate that the system development process be divided into phases and that certain documentation be produced by the end of each phase so that progress can be monitored and corrections made when necessary. These attempts; however, leave much to be desired and most organizations are attempting to improve the methods by which they manage their system development [20, 6].

This paper is concerned with one approach to improving systems development. The approach is based on three premises. The first is that more effort and attention should be devoted to the front end of the process where a proposed system is being described from the user's point of view [2, 14, 3]. The second premise is that the computer should be used in the development process since systems development involves large amounts of information processing. The third premise is that a computer-aided approach to systems development must start with "documentation."

This paper describes a computer-aided technique for documentation which consists of the following:

1) The results of each of the activities in the system development process are recorded in computer processible form as they are produced.

2) A computerized data base is used to maintain all the basic data about the system.

3) The computer is used to produce hard copy documentation when required.

The part of the technique which is now operational is known as PSL/PSA. Section II is devoted to a brief description of system development as a framework in which to compare manual and computer-aided documentation methods. The Problem Statement Language (PSL) is described in Section III. The reports which can be produced by the Problem Statement Analyzer (PSA) are described in Section IV. The status of the system, results of experience to date, and planned developments are outlined in Section V.

II. Logical systems design

The computer-aided documentation system described in Sections III and IV of this paper is designed to play an integral role during the initial stages in the system development process. A generalized model of the whole system development process is given in Section II-A. The final result of the initial stage is a document which here will be called the System Definition Report. The desired contents of this document are discussed in Section II-B. The activities required to produce this document manually are described in Section II-C and the changes possible through the use of computer-aided methods are outlined in Section II-D.

A. A model of the system development process

The basic steps in the life cycle of information systems (initiation, analysis, design, construction, test, installation, operation, and termination) appeared in the earliest applications of computers to organizational problems (see for example, [17, 1, 4, 7]). The need for more formal and comprehensive procedures for carrying out the life cycle was recognized; early examples are the IBM SOP publications [5], the Philips ARDI method [8], and the SDC method [23]. In the last few years, a large number of books and papers on this subject have been published [11, 19].

Examination of these and many other publications indicate that there is no general agreement on what phases the development process should be divided into, what documentation should be produced at each phase, what it should contain, or what form it should be presented in. Each organization develops its own methods and standards.

In this section a generalized system development process will be described as it might be conducted in an organization which has a Systems Department responsible for developing, operating, and maintaining computer based information processing systems. The System Department belongs to some higher unit in the organization and itself has some subunits, each with certain functions (see for example, [24]). The System Department has a system development standard procedure which includes a project management system and documentation standards.

A request for a new system is initiated by some unit in the organization or the system may be proposed by the System Department. An initial document is prepared which contains information about why a new system is needed and outlines its major functions. This document is reviewed and, if approved, a senior analyst is assigned to prepare a more detailed document. The analyst collects data by interviewing users and studying the present system. He then produces a report describing his proposed system and showing how it will satisfy the requirements. The report will also contain the implementation plan, benefit/cost analysis, and his recommendations. The report is reviewed by the various organizational units involved. If it passes this review it is then included with other requests for the resources of the System Department and given a priority. Up to this point the investment in the proposed system is relatively small.

At some point a project team is formed, a project leader and team members are assigned, and given authority to proceed with the development of the system. A steering group may also be formed. The project is assigned a schedule in accordance with the project management system and given a budget. The schedule will include one or more target dates. The final target date will be the date the system (or its first part if it is being done in parts) is to be operational. There may also be additional target dates such as beginning of system test, beginning of programming, etc.

B. *Logical system design documentation*

In this paper, it is assumed that the system development procedure requires that the proposed system be reviewed before a major investment is made in system construction. There will therefore be another target date at which the "logical" design of the proposed system is reviewed. On the basis of this review the decision may be to proceed with the physical design and construction, to revise the proposed system, or to terminate the project.

The review is usually based on a document prepared by the project team. Sometimes it may consist of more than one separate document; for example, in the systems development methodology used by the U.S. Department of Defense [21] for non-weapons systems, development of the life cycle is divided into phases. Two documents are produced at the end of the Definition subphase of the Development phase: a Functional Description, and a Data Requirements Document.

Examination of these and many documentation requirements show that a Systems Definition Report contains five major types of information:

1) a description of the organization and where the proposed system will fit; showing how the proposed system will improve the functioning of the organization or otherwise meet the needs which lead to the project;

2) a description of the operation of the proposed system in sufficient detail to allow the users to verify that it will in fact accomplish its objectives, and to serve as the specification for the design and construction of the proposed system if the project continuation is authorized;

3) a description of its proposed system implementation in sufficient detail to estimate the time and cost required;

4) the implementation plan in sufficient detail to estimate the cost of the proposed system and the time it will be available;

5) a benefit/cost analysis and recommendations.

In addition, the report usually also contains other miscellaneous information such as glossaries, etc.

C. Current logical system design process

During the initial stages of the project the efforts of the team are directed towards producing the Systems Definition Report. Since the major item this report contains is the description of the proposed system from the user or logical point of view, the activities required to produce the report are called the logical system design process. The project team will start with the information already available and then perform a set of activities. These may be grouped into five major categories.

1) *Data collection.* Information about the information flow in the present system, user desires for new information, potential new system organization, etc., is collected and recorded.

2) *Analysis.* The data that have been collected are summarized and analyzed. Errors, omissions, and ambiguities are identified and corrected. Redundancies are identified. The results are prepared for review by appropriate groups.

3) *Logical design.* Functions to be performed by the system are selected. Alternatives for a new system or modification of the present system are developed and examined. The "new" system is described.

4) *Evaluation.* The benefits and costs of the proposed system are determined to a suitable level of accuracy. The operational and functional feasibility of the system are examined and evaluated.

5) *Improvements.* Usually as a result of the evaluation a number of deficiencies in the proposed system will be discovered. Alternatives for improvement are identified and evaluated until further possible improvements are not judged to be worth additional effort. If major changes are made, the evaluation step may be repeated; further data collection and analysis may also be necessary.

In practice the type of activities outlined above may not be clearly distinguished and may be carried out in parallel or iteratively with increasing level of detail. Throughout the process, however it is carried out, results are recorded and documented.

It is widely accepted that documentation is a weak link in system development in general and in logical system design in particular. The representation in the documentation that is produced with present manual methods is limited to:

1) text in a natural language;

2) lists, tables, arrays, cross references;

3) graphical representation, figures, flowcharts.

Analysis of two reports showed the following number of pages for each type of information.

Form	Report A	Report B
text	90	117
lists and tables	207	165
charts and figures	28	54
total	335	336

The systems being documented are very complex, and these methods of representation are not capable of adequately describing all the necessary aspects of a system for all those who must, or should, use the documentation. Consequently, documentation is

1) ambiguous: natural languages are not precise enough to describe systems and different readers may interpret a sentence in different ways;

2) inconsistent: since systems are large the documentation is large and it is very difficult to ensure that the documentation is consistent;

3) incomplete: there is usually not a sufficient amount of time to devote to documentation and with a large complex system it is difficult to determine what information is missing.

The deficiencies of manual documentation are compounded by the fact that systems are continually changing and it is very difficult to keep the documentation up-to-date.

Recently there have been attempts to improve manual documentation by developing more formal methodologies [16, 12, 13, 22, 15, 25]. These methods, even though they are designed to be used manually, have a formal language or representation scheme that is designed to alleviate the difficulties listed above. To make the documentation more useful for human beings, many of these methods use a graphical language.

D. Computer-aided logical system design process

In computer-aided logical system design the objective, as in the manual process, is to produce the System Definition Report and the process followed is essentially similar to that described above. The computer-aided design system has the following capabilities:

1) capability to describe information systems, whether manual or computerized, whether existing or proposed, regardless of application area;

2) ability to record such description in a computerized data base;

3) ability to incrementally add to, modify, or delete from the description in the data base;

4) ability to produce "hard copy" documentation for use by the analyst or the other users.

The capability to describe systems in computer processible form results from the use of the system description language called PSL. The ability to record such description in a data base, incrementally modify it, and on demand perform analysis and produce reports comes from the software package called the Problem Statement Analyzer (PSA). The Analyzer is controlled by a Command Language which is described in detail in [9] (Fig. 1).

The Problem Statement Language is outlined in Section III and described in detail in [10]. The use of PSL/PSA in computer-aided logical system design is described in detail in [18].

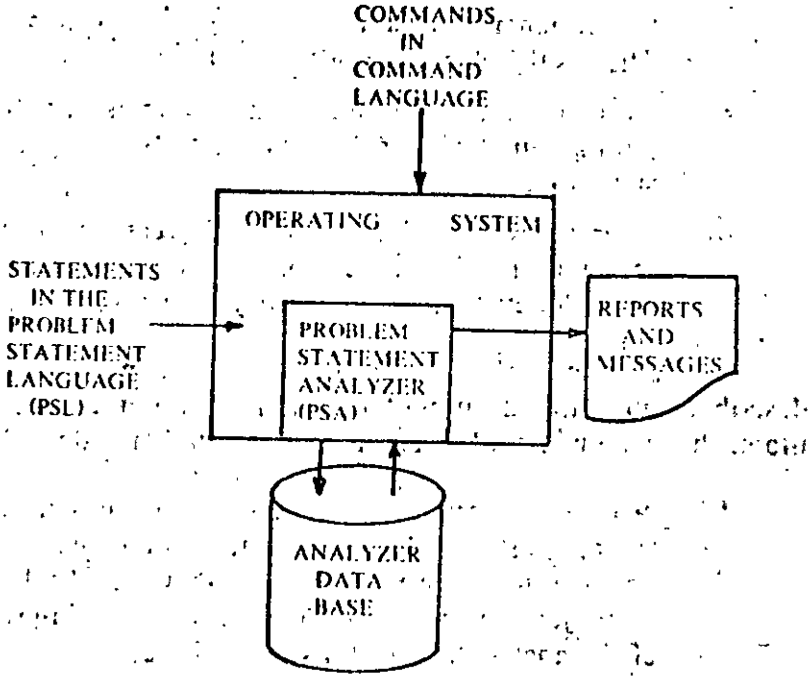

COMMANDS IN COMMAND LANGUAGE
OPERATING SYSTEM
STATEMENTS IN THE PROBLEM STATEMENT LANGUAGE (PSL)
PROBLEM STATEMENT ANALYZER (PSA)
REPORTS AND MESSAGES
ANALYZER DATA BASE

Figure 1. The problem statement analyzer.

The use of PSL/PSA does not depend on any particular structure of the system development process or any standards on the format and content of hard copy documentation. It is therefore fully compatible with current procedures in most organizations that are developing and maintaining systems. Using this system, the data collected or developed during all five of the activities are recorded in machine-readable form and entered into the computer as it is collected. A data base is built during the process. These data can be analyzed by computer programs and intermediate documentation prepared on request. The Systems Definition Report then includes a large amount of material produced automatically from the data base.

The activities in logical system design are modified when PSL/PSA is used as follows:

1) Data collection: since most of the data must be obtained through personal contact, interviews will still be required. The data collected are recorded in machine-readable form. The intermediate outputs of PSA also provide convenient checklists for deciding what additional information is needed and for recording it for input.

2) Analysis: a number of different kinds of analysis can be performed on demand by PSA, and therefore need no longer be done manually.

3) Design: design is essentially a creative process and cannot be automated. However, PSA can make more data available to the designer and allow him to manipulate it more extensively. The results of his decisions are also entered into the data base.

4) Evaluation: PSA provides some rudimentary facilities for computing volume or work measures from the data in the problem statement.

5) Improvements: identification of areas for possible improvements is also a creative task; however, PSA output, particularly from the evaluation phase, may be useful to the analyst.

The System Definition Report will contain the same material as that described since the documentation must serve the same purpose. Furthermore, the same general format and representation is desirable.

1) Narrative information is necessary for human readability. This is stored as part of the data but is not analyzed by the computer program. However, the fact that it is displayed next to, or in conjunction with, the final description improves the ability of the analyst to detect discrepancies and inconsistencies.

2) Lists, tables, arrays, matrices: These representations are prepared from the data base. They are up-to-date and can be more easily rearranged in any desired order.

3) Diagrams and charts. The information from the data base can be represented in various graphical forms to display the relationships between objects.

III. PSL, a problem statement language

PSL is a language for describing systems. Since it is intended to be used to describe "proposed" systems it was called a Problem Statement Language because the description of a proposed system can be considered a "problem" to be solved by the system designers and implementors.

PSL is intended to be used in situations in which analysts now describe systems. The descriptions of systems produced using PSL are used for the same purpose as that produced manually. PSL may be used both in batch and interactive environments, and therefore only "basic" information about the system need to be stated in PSL. All "derived" information can be produced in hard copy form as required.

The model on which PSL is based is described in Section III-A. A general description of the system and semantics of PSL is then given in Section III-B to illustrate the broad scope of system aspects that can be described using PSL. The detailed syntax of PSL in given in [10].

A. *Model of information systems*

The Problem Statement Language is based first on a model of a general system, and secondly on the specialization of the model to a particular class of systems, namely information systems.[1]

The model of a general system is relatively simple. It merely states that a system consists of things which are called OBJECTS. These objects may have PROPERTIES and each of these PROPERTIES may have PROPERTY VALUES. The objects may be connected or interrelated in various ways. These connections are called RELATIONSHIPS.[2]

The general model is specialized for an information system by allowing the use of only a limited number of predefined objects, properties, and relationships.

B. *An overview of the problem statement language syntax and semantics*

The objective of PSL is to be able to express in syntactically analyzable form as much of the information which commonly appears in System Definition Reports as possible.

System Descriptions may be divided into eight major aspects:

1) System Input/Output Flow,
2) System Structure,
3) Data Structure,
4) Data Derivation,
5) System Size and Volume,
6) System Dynamics,
7) System Properties,
8) Project Management.

PSL contains a number of types of objects and relationships which permit these different aspects to be described.

The *System Input/Output Flow* aspect of the system deals with the interaction between the target system and its environment.

System Structure is concerned with the hierarchies among objects in a system. Structures may also be introduced to facilitate a particular design approach such as "top down." All information may initially be grouped together and called by one name at the highest level, and then successively subdivided. System structures can represent high-level hierarchies which may not actually exist in the system, as well as those that do.

The *Data Structure* aspect of system description includes all the relationships which exist among data used and/or manipulated by the system as seen by the "users" of the system.

The *Data Derivation* aspect of the system description specifies which data objects are involved in particular PROCESSES in the system. It is concerned with what information is used, updated, and/or derived, how this is done, and by which processes.

Data Derivation relationships are internal in the system, while System Input/Output Flow relationships describe the system boundaries. As with other PSL facilities System Input/Output Flow need not be used. A system can be considered as having no boundary.

The *System Size and Volume* aspect is concerned with the size of the system and those factors which influence the volume of processing which will be required.

The *System Dynamics* aspect of system description presents the manner in which the target system "behaves" over time.

All objects (of a particular type) used to describe the target system have characteristics which distinguish them from other objects of the same type. Therefore, the PROPERTIES of particular objects in the system must be described. The PROPERTIES themselves are objects and given unique names.

The *Project Management* aspect requires that, in addition to the description of the target system being designed, documentation of the project designing (or documenting) the target system be given. This involves identification of people involved and their responsibilities, schedules, etc.

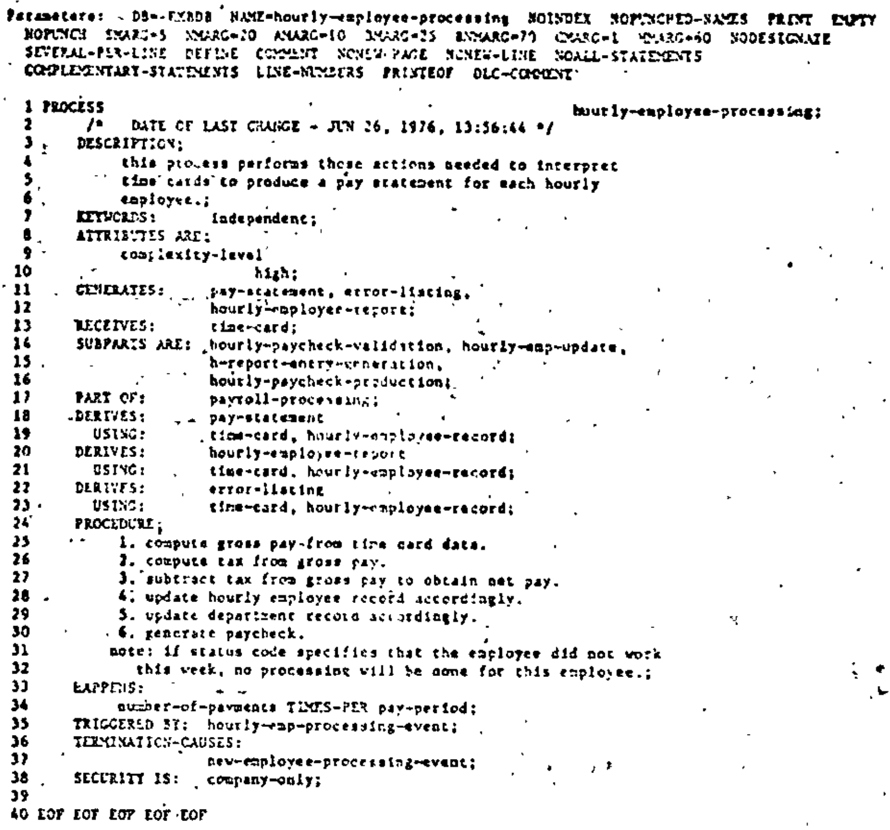
IV. Reports

As information about a particular system is obtained, it is expressed in PSL and entered into a data base using the Problem Statement Analyzer. At any time standard outputs or reports may be produced on request. The various reports can be classified on the basis of the purposes which they serve.

1) *Data Base Modification Reports:* These constitute a record of changes that have been made, together with diagnostics and warnings. They constitute a record of changes for error correction and recovery.

2) *Reference Reports:* These present the information in the data base in various formats. For example, the Name List Report presents all the objects in the data base with their type and date of last change. The Formatted Problem Statement Report shows all properties and relationships for a particular object

(Fig. 2). The Dictionary Report gives only data dictionary type information.



```
Parameters: DB=-EXBDB NAME=hourly-employee-processing NOINDEX NOPUNCHED-NAMES PRINT EMPTY
  NOPUNCH SMARG=5 NMARG=20 AMARG=10 DMARG=25 RNMARG=70 CMARG=1 HMARG=60 NODESIGNATE
  SEVERAL-PER-LINE DEFINE COMMENT NONEW-PAGE NONEW-LINE NOALL-STATEMENTS
  COMPLEMENTARY-STATEMENTS LINE-NUMBERS PRINTEOF DLC-COMMENT

PROCESS                                                  hourly-employee-processing;
     /*   DATE OF LAST CHANGE - JUN 26, 1976, 13:56:44 */
   DESCRIPTION;
        this process performs those actions needed to interpret
        time cards to produce a pay statement for each hourly
        employee.;
   KEYWORDS:      independent;
   ATTRIBUTES ARE:
        complexity-level
                       high;
   GENERATES:     pay-statement, error-listing,
                  hourly-employee-report;
   RECEIVES:      time-card;
   SUBPARTS ARE:  hourly-paycheck-validation, hourly-emp-update,
                  h-report-entry-generation,
                  hourly-paycheck-production;
   PART OF:       payroll-processing;
   DERIVES:       pay-statement
     USING:       time-card, hourly-employee-record;
   DERIVES:       hourly-employee-report
     USING:       time-card, hourly-employee-record;
   DERIVES:       error-listing
     USING:       time-card, hourly-employee-record;
   PROCEDURE;
        1. compute gross pay from time card data.
        2. compute tax from gross pay.
        3. subtract tax from gross pay to obtain net pay.
        4. update hourly employee record accordingly.
        5. update department record accordingly.
        6. generate paycheck.
       note: if status code specifies that the employee did not work
          this week, no processing will be done for this employee.;
   HAPPENS:
        number-of-payments TIMES-PER pay-period;
   TRIGGERED BY:  hourly-emp-processing-event;
   TERMINATION-CAUSES:
                  new-employee-processing-event;
   SECURITY IS:   company-only;

EOF EOF EOF EOF EOF
```

Figure 2. Example of a FORMATTED PROBLEM STATEMENT for one PROCESS.

3) *Summary Reports:* These present collections of information in summary from, or gathered from several different relationships. For example, the Data Base Summary Report provides project management information by showing the totals of various types of objects and how much has been said about them. The Structure Report shows complete or partial hierarchies. The Extended Picture Report shows the data flows in a graphical form.

4) *Analysis Reports:* These provide various types of analysis of the information in the data base. For example, the Contents Comparison Report analyzes similarity of Inputs and Outputs. The Data Process Interaction Report (Fig. 3) can be used to detect gaps in the information flow, or unused data objects. The Process Chain Report shows the dynamic behavior of the system (Fig. 4).

```
             11111111112222222222333
    12345678901234567890123456789012
   +----+----+----+----+----+----+--+
 1 :D   :    :    :    :    :    :  :
 2 : FDFFR   :    :    :    :    :  :
 3 :    : D  :    :    :    :    :  :
 4 :    :  R :    :    :    :    :  :
 5 +----+---R+----+----+----+----+--+
 6 :    :    R    :    :    :    :  :
 7 :    :    :DRR :    :    :    :  :
 8 :    :    :D   :    :    :    :  :
 9 :    :  F :   D:    :    :    :  :
10 +----+----+-F--+----+----+----+--+
11 :    :    :    R    :    :    :  :
12 :    :    :    DR   :    :    :  :
13 :    :    :    :DR  :    :    :  :
14 :    D    :    :R F :    :    :  :
15 +R---+----+----+----+----+----+--+
16 :  FFR    :    :    D    :    :  :
17 :D   :    :    :    :    :    :  :
18 :    :F   :    :    :FDFFR    :  :
19 :    :    :    :  RD:    :    :  :
20 +----+F---+----+----+---FDR---+--+
21 :    FD   :    :   R:    :    :  :
22 : FR :    :    :    :F  F:D   :  :
23 :    :    :R F :    :    :D   :  :
24 :    :   F:    :    :    : D  :  :
25 :    :    :    :    :    :   FF  :
   +----+----+----+----+----+----+--+
```

Figure 3. Example of part of a Data Process Interaction Report.

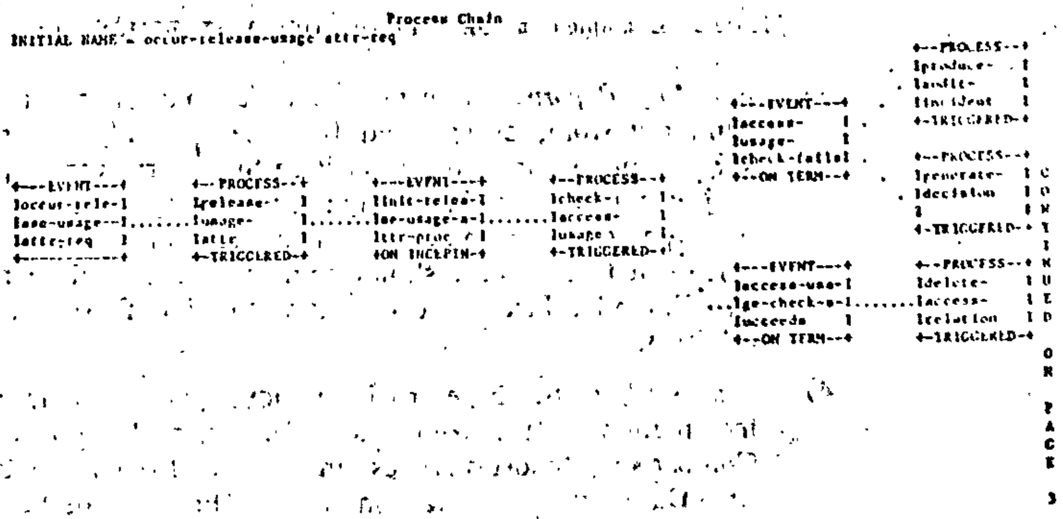


Figure 4. Example of a Process Chain Report.

After the requirements have been completed, the final documentation required by the organization can be produced semiautomatically to a presented format; e.g., the format required for the Functional Description and Data Requirements in [21].

V. Concluding remarks

The current status of PSL/PSA is described briefly in Section V-A. The benefits that should accrue to users of PSL/PSA are discussed in Section V-B. The information on benefits actually obtained by users is given in Section V-C. Planned extensions are outlined in Section V-D. Some conclusions reached as a result of the developments to date are given in Section V-E.

A. Current status

The PSL/PSA system described in this paper is operational on most larger computing environments which support interactive use, including IBM 370 series (OS/VS/TSO/CMS), Univac 1100 series (EXEC-8), CDC 6000/7000 series (SCOPE, TSS), Honeywell 600/6000 series (MULTICS, GCOS), AMDAHL 470/VS (MTS), and PDP-10 (TOPS 10). Portability is achieved at a relatively high level; almost all of the system is written in ANSI Fortran.

PSL/PSA is currently being used by a number of organizations including AT&T Long Lines, Chase Manhattan Bank, Mobil Oil, British Railways, Petroleos Mexicanos, TRW Inc., the U.S. Air Force and others for documenting systems. It is also being used by academic institutions for education and research.

B. Benefit/cost analysis of computer-aided documentation

The major benefits claimed for computer-aided documentation are that the "quality" of the documentation is improved and that the cost of design, implementation, and maintenance will be reduced. The "quality" of the documentation, measured in terms of preciseness, consistency, and completeness is increased because the analysts must be more precise, the software performs checking, and the output reports can be reviewed for remaining ambiguities, inconsistencies, and omissions. While completeness can never be fully guaranteed, one important feature of the computer-aided method is that all the documentation that "exists" is the data base, and therefore the gaps and omissions are more obvious. Consequently, the organization knows what data it has, and does not have to depend on individuals who may not be available when a specific item of data about a system is needed. Any analysis performed and reports produced are up-to-date as of the time it is performed. The coordination among analysts is greatly simplified since each can work in his own area and still have the system specifications be consistent.

Development will take less time and cost less because errors, which usually are not discovered until programming or testing, have been minimized. It is recognized that one reason for the high cost of systems development is the fact that errors, inconsistencies, and omissions in specifications are frequently not detected until later stages of development: in design, programming, systems tests, or even operation. The use of PSL/PSA during the specification stage reduces the number of errors which will have to be corrected later. Maintenance costs are considerably reduced because the effect of a proposed change can easily be isolated,, thereby reducing the probability that one correction will cause other errors.

The cost of using a computer-aided method during logical system design must be compared with the cost of performing the operations manually. In practice the cost of the various analyst functions of interviewing, recording, analyzing, etc., are not recorded separately. However, it can be argued that direct cost of documenting specifications for a proposed system using PSL/PSA should be approximately equal to the cost of producing the documentation manually. The cost of typing manual documentation is roughly equal to the cost of entering PSL statements into the computer. The computer cost of using PSA should not be more than the cost of analyst time in carrying out the analyses manually. (Computer costs, however, are much more visible than analysts costs.) Even though the total cost of logical system design is not reduced by using computer-aided methods, the elapsed time should be reduced because the computer can perform clerical tasks in a shorter time than analysts require.

C. *Benefits/costs evaluation in practice*

Ideally the adoption of a new methodology such as that represented by PSL/PSA should be based on quantitative evaluation of the benefits and costs. In practice this is seldom possible; PSL/PSA is no exception.

Very little quantitative information about the experience in using PSL/PSA, especially concerning manpower requirements and system development costs, is available. One reason for this lack of data is that the project has been concerned with developing the methodology and has not felt it necessary or worthwhile to invest resources in carrying out controlled experiments which would attempt to quantify the benefits. Furthermore, commercial and government organizations which have investigated PSL/PSA have, in some cases, started to use it without a formal evaluation; in other cases, they have started with an evaluation project. However, once the evaluation project is completed and the decision is made to use the PSL/PSA, there is little time or motivation to document the reasons in detail.

Organizations carrying out evaluations normally do not have the comparable data for present methods available and so far none have felt it necessary to run controlled experiments with both methods being used in parallel. Even when evaluations are made, the results have not been made available to the project, because the organizations regard the data as proprietary.

The evidence that the PSL/PSA is worthwhile is that almost without exception the organizations which have seriously considered using it have decided to adopt it either with or without an evaluation. Furthermore, practically all organizations which started to use PSL/PSA are continuing their use (the exceptions have been caused by factors other than PSL/PSA itself) and in organizations which have adopted it, usage has increased.

D. Planned developments

PSL as a system description language was intended to be "complete" in that the logical view of a proposed information system could be described, i.e., all the information necessary for functional requirements and specifications could be stated. On the other hand, the language should not be so complicated that it would be difficult for analysts to use. Also, deliberately omitted from the language was any ability to provide procedural "code" so that analysts would be encouraged to concentrate on the requirements rather than on low-level flow charts. It is clear, however, that PSL must be extended to include more precise statements about logical and procedural information.

Probably the most important improvement in PSA is to make it easier to use. This includes providing more effective and simple data entry and modification commands and providing more help to the users. A second major consideration is performance. As the data base grows in size and the number of users increases, performance becomes more important. Performance is very heavily influenced by factors in the computing environment which are outside the control of PSA development. Nevertheless, there are improvements that can be made.

PSL/PSA is clearly only one step in using computer-aided methods in developing, operating, and maintaining information processing systems. The results achieved to date support the premise that the same general approach can successfully be applied to the rest of the system life cycle and that the data base concept can be used to document the results of the other activities in the system life cycle. The resulting data bases can be the basis for development of methodology, generalized systems, education, and research.

E. Conclusions

The conclusions reached from the development of PSL/PSA to date and from the effort in having it used operationally may be grouped into five major categories.

e to the

hout ex-
decided
ly all or-
e excep-
rganiza-

lete" in
ed, i.e.,
fications
nplicated
ed from
analysts
on low-
include

asier to
ry and
d major
number
is very
outside
nts that

hods in
. The
ich can
ta base
he sys-
ent of

te and
major

1) The determination and documentation of requirements and functional specifications can be improved by making use of the computer for recording and analyzing the collected data and statements about the proposed system.

2) Computer-aided documentation is itself a system of which the software is only a part. If the system is to be used, adequate attention must be given to the whole methodology, including: user documentation, logistics and mechanics of use, training, methodological support, and management encouragement.

3) The basic structure of PSL and PSA is correct. A system description language should be of the relational type in which a description consists of identifying and naming objects and relationships among them. The software system should be database oriented, i.e., the data entry and modification procedures should be separated from the output report and analysis facilities.

4) The approach followed in the ISDOS project has succeeded in bringing PSL/PSA into operational use. The same approach can be applied to the rest of the system life cycle. A particularly important part of this approach is to concentrate first on the documentation and then on the methodology.

5) The decision to use a computer-aided documentation method is only partly influenced by the capabilities of the system. Much more important are factors relating to the organization itself and system development procedures. Therefore, even though computer-aided documentation is operational in some organizations, that does not mean that all organizations are ready to immediately adopt it as part of their system life cycle methodology.

References

1. T. Aiken, "Initiating an Electronics Program," in *Proceedings of the 7th Annual Meeting of the Systems and Procedures Association* (1954).
2. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19. No. 5 (May 1973), pp. 48-59.
3. ———, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," *Information Processing* (1974), pp. 192-97.
4. R.G. Canning, *Electronic Data Processing for Business and Industry* (New York: Wiley, 1956).
5. T.B. Glans, et al., *Management Systems* (New York: Holt, Rinehart, & Winston, 1968). Based on IBM's study Organization Plan, 1961.
6. J. Goldberg, ed., *Proceedings of the Symposium on High Cost of Software* (Stanford, Calif.: Stanford Research Institute, September 1973).
7. R.H. Gregory and R.L. Van Horn, *Automatic Data Processing Systems* (Belmont, Calif.: Wadsworth Publishing Co., 1960).
8. W. Hartman, H. Matthes, and A. Proeme, *Management Information Systems Handbook* (New York: McGraw-Hill, 1968).
9. E.A. Hershey and M. Bastarache, "PSA — Command Descriptions," University of Michigan, ISDOS Working Paper No. 91 (Ann Arbor, Mich.: 1975).
10. E.A. Hershey, et al., "Problem Statement Language — Language Reference Manual," University of Michigan, ISDOS Working Paper No. 68 (Ann Arbor, Mich.: 1975).
11. G.F. Hice, W.S. Turner, and L.F. Cashwell, *System Development Methodology* (Amsterdam, The Netherlands: North-Holland Publishing Co., 1974).
12. *HIPO — A Design Aid and Documentation Technique*, IBM Corporation, Manual No. GC-20-1851 (White Plains, N.Y.: IBM Data Processing Division, October 1974).
13. M.N. Jones, "Using HIPO to Develop Functional Specifications," *Datamation*, Vol. 22, No. 3 (March 1976), pp. 112-25.
14. G.H. Larsen, "Software: Man in the Middle," *Datamation*, Vol. 19, No. 11 (November 1973), pp. 61-66.
15. G.J. Myers, *Reliable Software Through Composite Design* (New York: Petrocelli/Charter, 1975).

16. D.T. Ross and K.E. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1 (January 1977), pp. 41-48.

17. H.W. Schrimpf and C.W. Compton, "The First Business Feasibility Study in the Computer Field," *Computers and Automation,* Vol. 18, No. 1 (January 1969), pp. 48-53.

18. D. Teichroew and M. Bastarache, "PSL User's Manual," University of Michigan, ISDOS Working Paper No. 98 (Ann Arbor, Mich.: 1975).

19. *Software Development and Configuration Management Manual,* TRW Systems Group, Manual No. TRW-55-73-07 (Redondo Beach, Calif.: December 1973).

20. U.S. Air Force, *Support of Air Force Automatic Data Processing Requirements Through the 1980's,* Electronics Systems Division, L.G. Hanscom Field, Report SADPR-85 (June 1974).

21. U.S. Department of Defense, *Automated Data Systems Documentation Standards Manual,* Manual 4120.17M (December 1972).

22. J.D. Warnier and B. Flanagan, *Entrainement de la Construction des Programs D'Informatique,* Vols. I and II (Paris: Editions d'Organization, 1972).

23. N.E. Willworth, ed., *System Programming Management,* System Development Corporation, TM 1578/000/00 (Santa Monica, Calif.: March 13, 1964).

24. F.G. Withington, *The Organization of the Data Processing Function* (New York: Wiley Business Data Processing Library, 1972).

25. E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, N.J.: Prentice-Hall, 1979). (1979 edition of YOURDON's 1975 text.)

21

INTRODUCTION

DeMarco's "Structured Analysis and System Specification" is the final paper chosen for inclusion in this book of classic articles on the structured revolution. It is last of three on the subject of analysis, and, together with Ross/Schoman [Paper 22] and Teichroew/Hershey [Paper 23], provides a good idea of the direction that structured analysis will be taking in the next few years.

Any competent systems analyst undoubtedly could produce a five-page essay on "What's Wrong with Conventional Analysis." DeMarco, being an ex-analyst, does so with pithy remarks, describing conventional analysis as follows:

> "Instead of a meaningful interaction between analyst and user, there is often a period of fencing followed by the two parties' studiously ignoring each other. . . The cost-benefit study is performed backwards by deriving the development budget as a function of expected savings. (Expected savings were calculated by prorating cost reduction targets handed down from On High.)"

In addition to providing refreshing prose, DeMarco's approach differs somewhat — in terms of emphasis — from that of Teichroew/Hershey and of Ross/Schoman. Unlike his colleagues, DeMarco stresses the importance of the *maintainability* of the specification. Take, for instance, the case of one system consisting of six *million* lines of COBOL and written over a period of ten years by employees no longer with the organization. Today, *nobody knows what the system does!* Not only have the program listings and source code been lost — a relatively minor disaster that we all have seen too often — but the specifications are completely out of date. Moreover, the system has grown so large that neither the users nor the data processing people have the faintest idea of *what* the system is supposed to be doing, let alone *how* the mysterious job is being accomplished! The example is far from hypothetical, for this is the fate that all large systems eventually will suffer, unless steps are taken to keep the *specifications* both current and understandable across generations of users.

The approach that DeMarco suggests — an approach generally known today as structured analysis — is similar in form to that proposed by Ross and Schoman, and emphasizes a top-down, partitioned, graphic model of the system-to-be. However, in contrast to Ross and Schoman, DeMarco also stresses the important role of a *data dictionary* and the role of scaled-down specifications, or mini-specs, to be written in a rigorous subset of the English language known as *Structured English*.

DeMarco also explains carefully how the analyst proceeds from a physical description of the user's current system, through a logical description of that same system, and eventually into a logical description of the new system that the user wants. Interestingly, DeMarco uses top-down, partitioned dataflow diagrams to illustrate this part of the so-called Project Life Cycle — thus confirming that such a graphic model can be used to portray virtually any system.

As in other short papers on the subject, the details necessary for carrying out DeMarco's approach are missing or are dealt with in a superficial manner. Fortunately, the details *can* be found: Listed at the end of the paper are references to three full-length books and one videotape training course, all dealing with the kind of analysis approach recommended by DeMarco.

DEMARCO 24

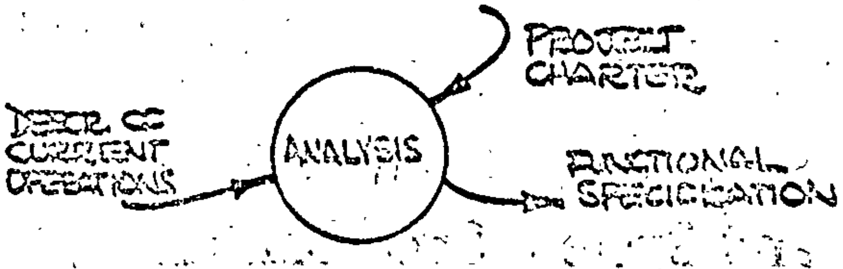
Structured Analysis and System Specification

When Ed Yourdon first coined the term Structured Analysis [1], the idea was largely speculative. He had no actual results of completed projects to report upon. His paper was based on the simple observation that some of the principles of top-down partitioning used by designers could be made applicable to the Analysis Phase. He reported some success in helping users to work out system details using the graphics characteristic of structured techniques.

Since that time, there has been a revolution in the methodology of analysis. More than 2000 companies have sent employees to Structured Analysis training seminars at YOURDON alone. There are numerous working texts and papers on the subject [2, 3, 4, 5]. In response to the YOURDON 1977 Productivity Survey [6, 7], more than one quarter of the respondents answered that they were making some use of Structured Analysis. The 1978 survey [8] shows a clear increase in that trend.

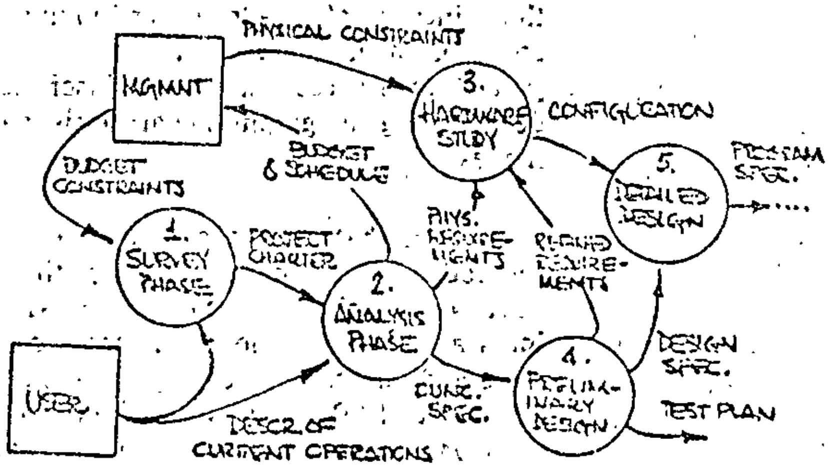
In this paper, I shall make a capsule presentation of the subject of Structured Analysis and its effect on the business of writing specifications of to-be-developed systems. I begin with a set of definitions:

1. What is analysis?



The analysis transformation.

Analysis is the process of transforming a stream of information about current operations and new requirements into some sort of rigorous description of a system to be built. That description is often called a Functional Specification or System Specification.



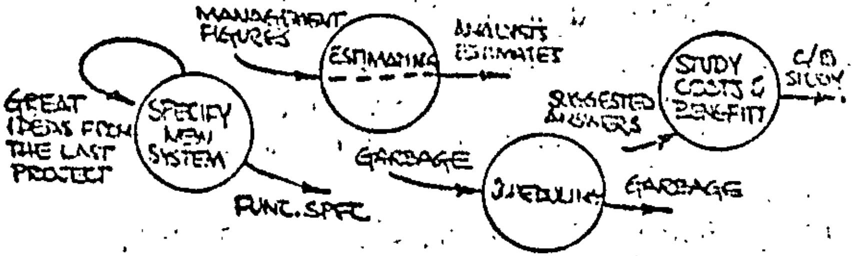
The Analysis Phase in context of the project life cycle.

In the context of the project life cycle for system development, analysis takes place near the beginning. It is preceded only by a Survey or Feasibility Study, during which a project charter (statement of changes to be considered, constraints governing development, etc.) is generated. The Analysis Phase is principally concerned with generating a specification of the system to be built.

But there are numerous required by-products of the phase, including budget, schedule and physical requirements information. The specification task is compounded of the following kinds of activities:

- user interaction,
- study of the current environment,
- negotiation,
- external design of the new system,
- I/O format design,
- cost-benefit study,
- specification writing, and
- estimating.

Well, that's how it works when it works. In practice, some of these activities are somewhat shortchanged. After all, everyone is eager to get on to the real work of the project (writing code). Instead of a meaningful interaction between analyst and user, there is often a period of fencing followed by the two parties' studiously ignoring each other. The study of current operations is frequently bypassed. ("We're going to change all that, anyway.") Many organizations have given up entirely on writing specifications. The cost-benefit study is performed backwards by deriving the development budget as a function of expected savings. (Expected savings were calculated by prorating cost reduction targets handed down from On High.) And the difficult estimating process can be conveniently replaced by simple regurgitation of management's proposed figures. So the Analysis Phase often turns out to be a set of largely disconnected and sometimes fictitious processes with little or no input from the user:



Analysis Phase activities.

2. What is Structured Analysis?

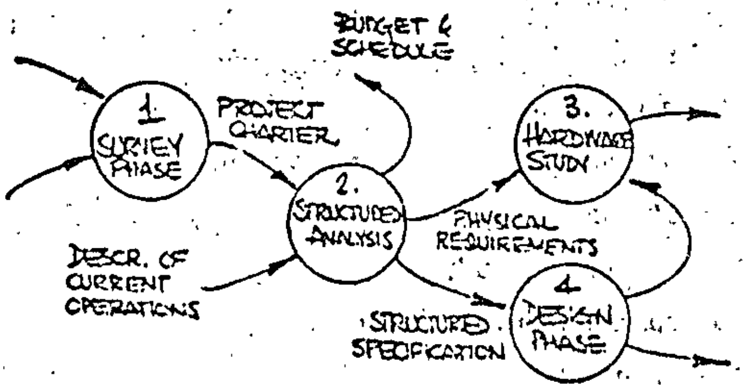


Figure 0: Structured Analysis in context of the project life cycle.

Structured Analysis is a modern discipline for conduct of the Analysis Phase. In the context of the project life cycle, its major apparent difference is its new principal product, called a Structured Specification. This new kind of specification has these characteristics:

- It is *graphic,* made up mostly of diagrams.
- It is *partitioned,* not a single specification, but a network of connected "mini-specifications."
- It is *top-down,* presented in a hierarchical fashion with a smooth progression from the most abstract upper level to the most detailed bottom level.
- It is *maintainable,* a specification that can be updated to reflect change in the requirement.
- It is *a paper model of the system-to-be;* the user can work with the model to perfect his vision of business operations as they will be with the new system in place.

I'll have more to say about the Structured Specification in a later section.

In the preceding figure, I have represented Structured Analysis as a single process (transformation of the project charter and description of current operations into outputs of budget, schedule, physical requirements and the Structured Specification). Let's now look at the details of that process:

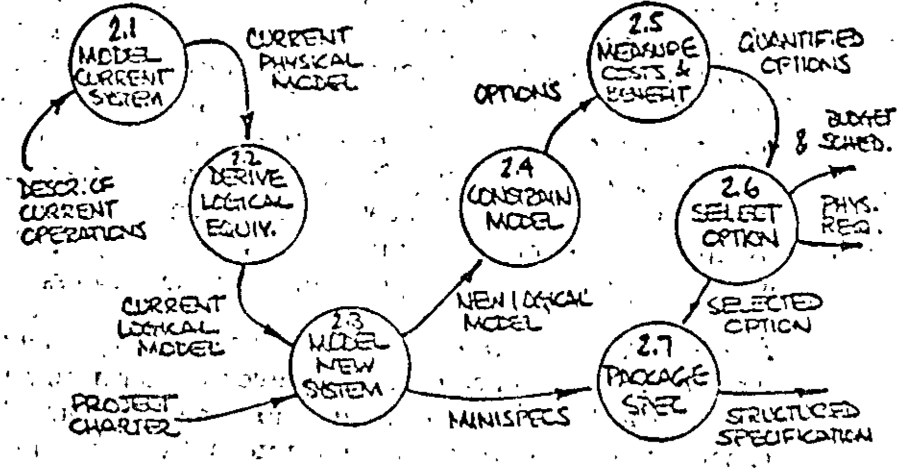


Figure 2: Details of Bubble 2, Structured Analysis.

Note that Figure 2 portrays exactly the same transformation as *Bubble 2* of the previous figure (same inputs, same outputs). But it shows that transformation in considerably more detail. It declares the component processes that make up the whole, as well as the information flows among them.

I won't insult your intelligence by giving you the thousand words that the picture in Figure 2 is worth. Rather than discuss it directly, I'll let it speak for itself. All that is required to complement the figure is a definition of the component processes and information flows that it declares:

> *Process 2.1. Model the current system:* There is almost always a current system (system = integrated set of manual and perhaps automated processes, used to accomplish some business aim). It is the environment that the new system will be dropped into. Structured Analysis would have us build a paper model of the current system, and use it to perfect our understanding of the present environment. I term this model "physical" in that it makes use of the user's terms, procedures, locations, personnel names and particular ways of carrying out business policy. The justification for the physical nature of this first model is that its purpose is to be a verifiable representation of current operations, so it must be easily understood by user staff.
>
> *Process 2.2. Derive logical equivalent:* The logical equivalent of the physical model is one that is divorced from the "hows" of the current operation. It concentrates instead on the "whats." In place of a description of a way to carry out policy, it strives to be a description of the policy itself.

Process 2.3. Model the new system: This is where the major work of the Analysis Phase, the "invention" of the new system, takes place. The project charter records the differences and potential differences between the current environment and the new. Using this charter and the logical model of the existing system, the analyst builds a new model, one that documents operations of the future environment (with the new system in place), just as the current model documents the present. The new model presents the system-to-be as a partitioned set of elemental processes. The details of these processes are now specified, one mini-spec per process.

Process 2.4. Constrain the model: The new logical model is too logical for our purposes, since it does not even establish how much of the declared work is done inside and how much outside the machine. (That is physical information, and thus not part of a logical model.) At this point, the analyst establishes the man-machine boundary, and hence the scope of automation. He/she typically does this more than once in order to create meaningful alternatives for the selection process. The physical considerations are added to the model as annotations.

Process 2.5. Measure costs and benefits: A cost-benefit study is now performed on each of the options. Each of the tentatively physicalized models, together with its associated cost-benefit parameters, is passed on in the guise of a "Quantified Option."

Process 2.6. Select option: The Quantified Options are now analyzed and one is selected as the best. The quanta associated with the option are formalized as budget, schedule, and physical requirements and are passed back to management.

Process 2.7. Package the specification: Now all the elements of the Structured Specification are assembled and packaged together. The result consists of the selected new physical model, the integrated set of mini-specs and perhaps some overhead (table of contents, short abstract, etc.).

3. What is a model?

The models that I have been referring to throughout are paper representations of systems. A system is a set of manual and automated procedures used to effect some business goal. In the convention of Structured Analysis, a model is made up of *Data Flow Diagrams* and a *Data Dictionary*. My definitions of these two terms follow:

A *Data Flow Diagram* is a network representation of a system. It presents the system in terms of its component processes, and declares all the interfaces among the components. All of the figures used so far in this paper have been Data Flow Diagrams.

A *Data Dictionary* is a set of definitions of interfaces declared on Data Flow Diagrams. It defines each of these interfaces (dataflows) in terms of its components. If I had supplied an entire model of the project life cycle (instead of just an incomplete Data Flow Diagram portion), you would turn to the Data Dictionary of that model to answer any questions that might have arisen in your study of Figure 0 and Figure 2 above. For instance, if you had puzzled over what the Data Flow Diagram referred to as a Quantified Option, you would look it up in the Data Dictionary. There you would find that Quantified Option was made up of the physicalized Data Flow Diagram, Data Dictionary, purchase cost of equipment, schedule and budget for development, monthly operating cost, risk factors, etc.

Data Flow Diagrams are often constructed in leveled sets. This allows a top-down partitioning: The system is divided into subsystems with a top-level Data Flow Diagram; the subsystems are divided into sub-subsystems with second-level Data Flow Diagrams, and so on. The Data Flow Diagrams describing the project life cycle were presented as part of a leveled set. Figure 0 was the parent (top of the hierarchy), and Figure 2, a child. If the model were complete, there would be other child figures as well, siblings of Figure 2. Figure 2 might have some children of its own (Figure 2.1, describing the process of modeling the current system in more detail; Figure 2.2, describing the derivation of logical equivalents, etc.).

The system model plays a number of different roles in Structured Analysis:

- *It is a communication tool.* Since user and analyst have a long-standing history of failure to communicate, it is essential that their discussions be conducted over some workable negotiating instrument, something to point to as they labor to reach a common understanding. Their discussion concerns systems, both past and present, so a useful system model is the most important aid to communication.

- *It is a framework for specification.* The model declares the component pieces of the system, and the pieces of those pieces, all the way down to the bottom. All that remains is to specify the bottom-level processes (those that are not further subdivided). This is accomplished by writing one mini-spec for each bottom-level process.

- *It is a starting point for design.* To the extent that the model is the most eloquent statement of requirement, it has a strong shaping influence on work of the Design Phase. To the extent that the resultant design reflects the shape of the model, it will be conceptually easy to understand for maintainers and user staff. The natural relationship between the Analysis Phase model and the design of the system (its internal structure) is akin to the idea that "form ever follows function."

So far, all I've done is define terms, terms relevant to the analysis process and to Structured Analysis in particular. With these terms defined, we can turn our attention to two special questions: What has been wrong with our approach to computer systems analysis in the past? and, How will Structured Analysis help?

4. What is wrong with classical analysis?

Without going into a long tirade against classical methods, the major problem of analysis has been this: Analysts and users have not managed to communicate well enough — the systems delivered too often have not been the systems the users wanted. Since analysis establishes development goals, failures of analysis set projects moving in wrong directions. The long-standing responses of management to all development difficulties (declaration of the 70-hour work week, etc.) do not help. They only goad a project into moving more quickly, still in the wrong direction. The more manpower and dedication added, the further the project will move away from its true goals. Progress made is just more work to be undone later. There is a word that aptly describes such a project, one that is set off in the wrong direction by early error and cannot be rescued by doubling and redoubling effort. The word is "doomed." Most of the Great Disasters of EDP were projects doomed by events and decisions of the Analysis Phase. All the energies and talents expended thereafter were for naught.

The major failures of classical analysis have been failures of the specification process. I cite these:

The monolithic approach. Classical Functional Specifications read like Victorian novels: heavy, dull and endless. No piece has any meaning by itself. The document can only be read serially, from front to back. No one ever reads the whole thing. (So no one reads the end.)

The poured-in-concrete effect. Functional Specifications are impossible to update. (I actually had one analyst tell me that the change we were considering could be more easily made to the system itself, when it was finally delivered, than to the specification.) Since they can't be updated, they are always out of date.

The lack of feedback. Since pieces of Functional Specifications are unintelligible by themselves, we have nothing to show the user until the end of analysis. Our author-reader cycle may be as much as a year. There is no possibility of iteration (that is, frequently repeated attempts to refine and perfect the product). For most of the Analysis Phase, there is no product to iterate. The famous user-analyst dialogue is no dialogue at all, but a series of monologues.

The Classical Functional Specification is an unmaintainable monolith, the result of "communication" without benefit of feedback. No wonder it is *neither functional nor specific.*

5. How does Structured Analysis help?

I would not be writing this if I did not believe strongly in the value of Structured Analysis; I have been known to wax loquacious on its many virtues. But in a few words, these are the main ways in which Structured Analysis helps to resolve Analysis Phase problems:

- It attacks the problem of largeness by *partitioning.*
- It attacks the many problems of communication between user and analyst by *iterative communication* and an *inversion of viewpoint.*
- It attacks the problem of specification maintenance by *limited redundancy.*

Since all of these ideas are rather new in their application to analysis, I provide some commentary on each:

The concept of *partitioning* or "functional decomposition" may be familiar to designers as the first step in creating a structured design. Its potential value in analysis was evident from the beginning — clearly, large systems cannot be analyzed without some form of concurrent partitioning. But the direct application of Design Phase functional decomposition tools (structure charts and HIPO) caused more problems than it solved. After some early successful experiments [9], negative user attitudes toward the use of hierarchies became apparent, and the approach was largely abandoned. Structured Analysis teaches use of leveled Data Flow Diagrams for partitioning, in place of the hierarchy. The advantages are several:

- The appearance of a Data Flow Diagram is not at all frightening. It seems to be simply a picture of the subject matter being discussed. You never have to explain an arbitrary convention to the user — you don't explain anything at all. You simply use the diagrams. I did precisely that with you in this paper; I used Data Flow Diagrams as a descriptive tool, long before I had even defined the term.

- Network models, like the ones we build with Data Flow Diagrams, are already familiar to some users. Users may have different names for the various tools used — Petri Networks or Paper Flow Charts or Document Flow Diagrams — but the concepts are similar.

- The act of partitioning with a Data Flow Diagram calls attention to the interfaces that result from the partitioning. I believe this is important, because the complexity of interfaces is a valuable indicator of the quality of the partitioning effort: the simpler the interfaces, the better the partitioning.

I use the term *iterative communication* to describe the rapid two-way interchange of information that is characteristic of the most productive work sessions. The user-analyst dialogue has got to be a dialogue. The period over which communication is turned around (called the author-reviewer cycle) needs to be reduced from months to minutes. I quite literally mean that fifteen minutes into the first meeting between user and analyst, there should be some feedback. If the task at hand requires the user to describe his current operation, then fifteen minutes into that session is not too early for the analyst to try telling the user what he has learned. "OK, let me see if I've got it right. I've drawn up this little picture of what you've just explained to me, and I'll explain it back to you. Stop me when I go wrong."

Of course, the early understanding is always imperfect. But a careful and precise declaration of an imperfect understanding is the best tool for refinement and correction. The most important early product on the way to developing a good product is an imperfect version. The human mind is an iterative processor. It never does anything precisely right the first time. What it does consummately well is to make a slight improvement to a flawed product. This it can do again and again. The idea of developing a flawed early version and then refining and refining to make it right is a very old one. It is called *engineering*. What I am proposing is an engineering approach to analysis and to the user-analyst interface.

The product that is iterated is the emerging system model. When the user first sees it, it is no more than a rough drawing made right in front of him during the discussion. At the end, it is an integral part of the Structured Specification. By the time he sees the final specification, each and every page of it should have been across his desk a half dozen times or more.

I mentioned that Structured Analysis calls for an *inversion of viewpoint*. This point may seem obscure because it is not at all obvious what the viewpoint of classical analysis has been. From my reading of hundreds of Classical Functional Specifications over the past fifteen years, I have come to the conclusio that their viewpoint is most often that of the computer. The classic specification describes what the computer does, in the order that it does it, u ing terms that are relevant to computers and computer people. It frequent limits itself to a discussion of processing inside the machine and da transferred in and out. Almost never does it specify anything that happens ou side the man-machine boundary.

The machine's viewpoint is natural and useful for those whose concer lie inside (the development staff), and totally foreign to those whose concer lie outside (the user and his staff). Structured Analysis adopts a differe viewpoint, that of the data. A Data Flow Diagram follows the data paths wh ever they lead, through manual as well as automated procedure. A correc drawn system model describes the outside as well as the inside of the automa ed portion. In fact, it describes the automated portion *in the context of the co plete system.* The viewpoint of the data has two important advantages over th of any of the processors:

- The data goes everywhere, so its view is all-inclusive.
- The viewpoint of the data is common to those concerned with the inside as well as those concerned with the outside of the man-machine boundary.

The use of *limited redundancy* to create a highly maintainable product not new. Any programmer worth his salt knows that a parameter that is lik to be changed ought to be defined in one place and one place only. What new in Structured Analysis is the idea that the specification ought to be ma tainable at all. After all, aren't we going to "freeze" it? If we have learn anything over the past twenty years, it is that the concept of freezi specifications is one of the great pipe dreams of our profession. Change cann be forestalled, only ignored. Ignoring change assures the building of a produ that is out of date and unacceptable to the user. We may endeavor to hold selected changes to avoid disruption of the development effort, but we can longer tolerate being *obliged* to ignore change just because our specification impossible to update. It is equally intolerable to accept the change, without u dating the specification. The specification is our mechanism for keeping t project on target, tracking a moving goal. Failure to keep the specification up date is like firing away at a moving target with your eyes closed. A key conc of Structured Analysis is development of a specification with little or no red dancy. This is a serious departure from the classical method that calls specification of everything at least eleven times in eleven places. Reduc redundancy, as a by-product, makes the resultant specification considera more concise.

6. What is a Structured Specification?

Structured Analysis is a discipline for conduct of the Analysis Phase. It includes procedures, techniques, documentation aids, logic and policy description tools, estimating heuristics, milestones, checkpoints and by-products. Some of these are important, and the rest merely convenient. What is most important is this simple idea: Structured Analysis involves building a new kind of specification, a Structured Specification, made up of Data Flow Diagrams, Data Dictionary and mini-specs.

The roles of the constituent parts of the Structured Specification are presented below:

> The *Data Flow Diagrams* serve to partition the system. The system that is treated by the Data Flow Diagram may include manual as well as automated parts, but the same partitioning tool is used throughout. The purpose of the Data Flow Diagram is not to specify, but to declare. It declares component processes that make up the whole, and it declares interfaces among the components. Where the target system is large, several successive partitionings may be required. This is accomplished by lower-level Data Flow Diagrams of finer and finer detail. All the levels are combined into a leveled DFD set.
>
> The *Data Dictionary* defines the interfaces that were declared on the Data Flow Diagrams. It does this with a notational convention that allows representation of dataflows and stores in terms of their components. (The components of a dataflow or store may be lower-level dataflows, or they may be data elements.) Before the Structured Specification can be called complete, there must be one definition in the Data Dictionary for each dataflow or data store declared on any of the Data Flow Diagrams.
>
> The *mini-specs* define the elemental processes declared on the Data Flow Diagrams. A process is considered elemental (or "primitive") when it is not further decomposed into a lower-level Data Flow Diagram. Before the Structured Specification can be called complete, there must be one mini-spec for each primitive process declared on any of the Data Flow Diagrams.
>
> The YOURDON Structured Analysis convention includes a set of rules and methods for writing mini-specs using Structured English, decision tables and trees and certain non-linguistic techniques for specification. For the purposes of this paper, such considerations are at the detail level — once you have partitioned to the point at which each of the mini-specs can be written in a page or less, it doesn't matter too terribly much how you write them.

7. What does it all mean?

Structured Analysis is here to stay. I estimate that its serious user con
munity now includes more than 800 companies worldwide. Users as far awa
as Australia and Norway have published results of the application of Structure
Analysis techniques to real-world projects [10, 11]. There are courses, tex
and even a video-tape series [12] on the subject. There are automated suppo
tools for use in Structured Analysis [13]. There are rival notations and symbo
ogies [3, 14, 15]. Working sessions have been cropping up at GUIDE, NCC
AMA and the DPMA. There are user groups, templates and T-shirts.

The fundamentals of Structured Analysis are not new. Most of the idea
have been used piecemeal for years. What is new is the emerging disciplin
The advantages of this discipline are substantial. They include a much mor
methodical approach to specification, a more usable and maintainable produc
and fewer surprises when the new system is installed. These are especially a
tractive when compared to the advantages of the classical discipline for analysi
There were no advantages. There was no discipline.

References

1. E. Yourdon, "The Emergence of Structured Analysis," *Computer Decisions*, Vol. 8, No. 4 (April 1976), pp. 58-59.
2. T. DeMarco, *Structured Analysis and System Specification* (New York: YOURDON Press, 1978).
3. C. Gane and T. Sarson, *Structured Systems Analysis* (New York: Improved System Technologies, Inc., 1977).
4. T. DeMarco, "Breaking the Language Barrier," *Computerworld*, Vol. XII, Nos. 32, 33 and 34 (August 7, 14 and 21, 1978) [published in parts].
5. *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977). Special issue on structured analysis.
6. *The YOURDON Report*, Vol. 2, No. 3 (March 1977).
7. T. DeMarco, "Report on the 1977 Productivity Survey" (New York: YOURDON inc., September 1977).
8. *The YOURDON Report*, Vol. 3, No. 3 (June-July 1978).
9. M. Jones, "Using HIPO to Develop Functional Specifications," *Datamation*, Vol. 22, No. 3 (March 1976), pp. 112-25.
10. J. Simpson, "Analysis and Design — A Case Study in a Structured Approach," *Australasian Computerworld*, Vol. 1, No. 2 (July 21, 1978), pp. 2-11, 13.
11. J. Pedersen and J. Buckle, "Kongsberg's Road to an Industrial Software Methodology," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 4 (July 1978).
12. *Structured Analysis*, Videotape series (Chicago: DELTAK inc., 1978).
13. D. Teichroew and E. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1 (January 1977), pp. 41-48.
14. *Introduction to SADT*, SofTech Inc., Document No. 9022-78 (Waltham, Mass.: February 1976).
15. V. Weinberg, *Structured Analysis* (New York: YOURDON Press, 1978).

37

INSIDE:
SADT Notation, Modeling, Design Representation, SADT vs. Other Design Methodologies

Software Design Using SADT

PAYOFF IDEA. Bertrand Russell has written that "a good notation has a subtlety and suggestiveness which at times makes it seem almost like a live teacher...a perfect notation would be a substitute for thought." Software designers need a conceptual and notational framework for expressing, evaluating, comparing, and choosing among design alternatives. SADT as a language and as a methodology directs and disciplines the analysis and design of systems. This portfolio surveys the process of defining a system using SADT.

INTRODUCTION

SADT™ (Structured Analysis and Design Technique) is bot graphic language for describing systems and a methodology for prod ing such descriptions. A system can be viewed as consisting of thi (objects, documents, or data). happenings (activities performed by p ple, machines, or software), and their interrelationships. This fun mental starting point accounts for SADT's broad applicability: it car used, for example, for the definition of system requirements (with necessarily specifying whether they are to be achieved by people. software, or by machines), a structured software design, and eve manual procedure.

The use of SADT for the definition of requirements has been scribed in other articles (see [2, 13, 14]). This portfolio focuses on features of the graphic language that provide a vocabulary well suite describing design. SADT also includes steps for ensuring adequate sign review and effective use of design walkthroughs: these methods similar to those described for use during analysis [2, 14].

™ SADT is a registered trademark of SofTech Inc

SADT was developed initially by Douglas T. Ross during the period from 1969 to 1973. It has been further developed and refined at SofTech as a result of extensive use since 1973. SADT has been sucessfully applied to a variety of complex system problems ranging from real-time telephonic communications design [16] and computer-aided manufacturing [18] to military policy planning [8]. The largest software system designed with SADT is a microprocessor-based PABX telephonic system developed by IT&T [10].

THE SADT NOTATION

An SADT system description is a set of diagrams, each depicting only a limited amount of detail. The diagrams expose a system structure, a piece at a time, from the top down. Quite literally, SADT supports the principle that "to divide is to conquer," provided we know how the divided pieces are combined to constitute the whole. This proviso applies to the diagram contents as well as to the relationships among diagrams. The rules of language usage encourage and even enforce unfolding a system description in intellectually manageable information units. At all times, the system structure and hence the relationship of any part to the whole is graphically visible.

The syntactic notation of SADT has been described in detail elsewhere [12]: diagrams consist principally of boxes (representing parts of a whole) interconnected by arrows (representing interfaces between the parts), each suitably labeled with nouns (for things) or verb phrases (for happenings). Each box on a diagram (representing, for example, a system activity) can be further detailed on a separate diagram with more interconnected boxes and arrows. The notation requires that a box and the corresponding diagram detailing it represent exactly the same part of a system. In particular, the external interfaces in the form of inputs, outputs, and controls must match.

Given the preceding characteristics, the diagrams in an SADT system description can be placed directly into a tree-like hierarchy; indexing information is encoded so that the appropriate context for any diagram can be immediately determined. Figure 1 depicts two diagrams; a box on the first diagram is detailed by the second diagram.

The features of any language encourage certain patterns of thought. With SADT, the inputs, outputs, and controls of the system under study are determined first. The purpose and viewpoint for the analysis are then identified. This logical progression focuses the effort so that extraneous considerations can be avoided.

Modeling from a Viewpoint for a Purpose

Often, multiple models of a system from differing viewpoints are required for an adequate understanding. An SADT model can provide a context for other relevant material such as explanatory text, supporting documents, and forms. An SADT design model can thus actually be a group of models, each representing an important design decision that was isolated. Figure 2 shows two models (i.e., system descriptions with different viewpoints) sharing common detail. Consistency can be

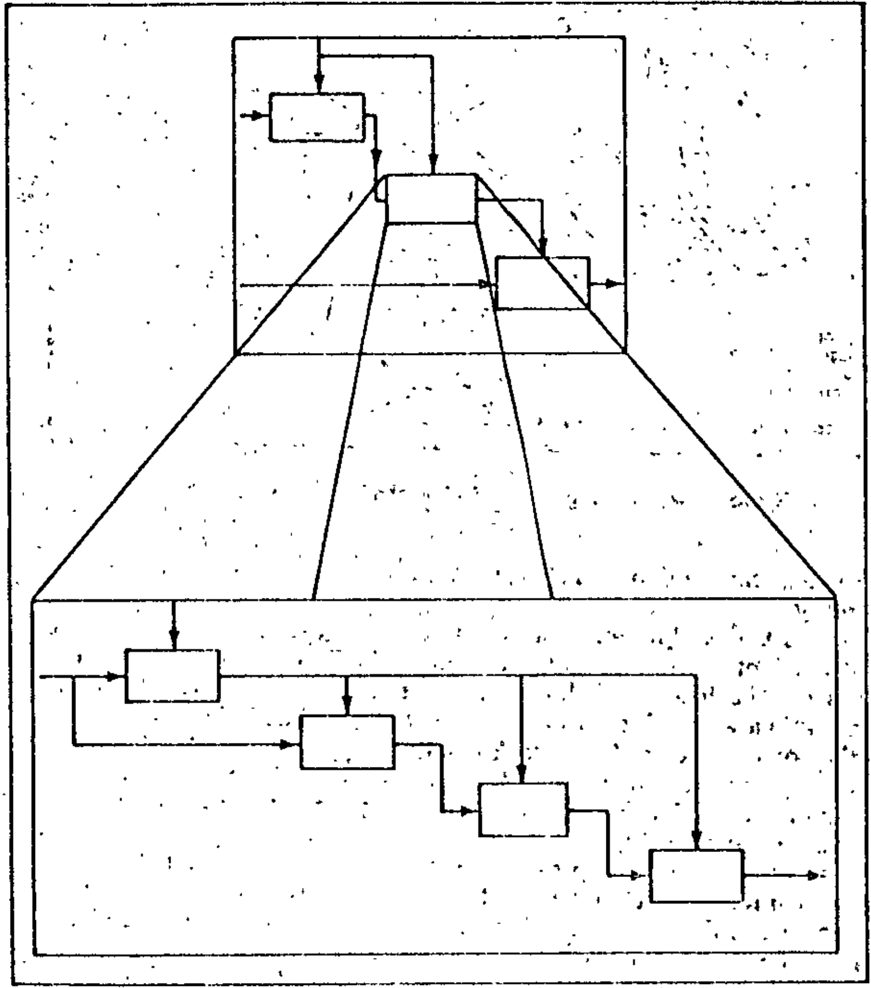


Figure 1. A Diagram Detailing the Box of Another

checked by "tying" together the different models for a system, that is, by making their interrelationships explicit.

REPRESENTING A DESIGN WITH SADT

SADT does not assume the designer is using one particular design methodology. The relationship of SADT to several widely used design methodologies is briefly discussed later in this portfolio.

Mechanism

An important aspect of the design vocabulary is the concept of an SADT mechanism. A mechanism is a distinct model that has been isolated for reasons of shareability and changeability. As such, mechanisms are naturally suited to the information-hiding approaches (e.g., abstract data types, monitors, and levels of abstraction). Major design

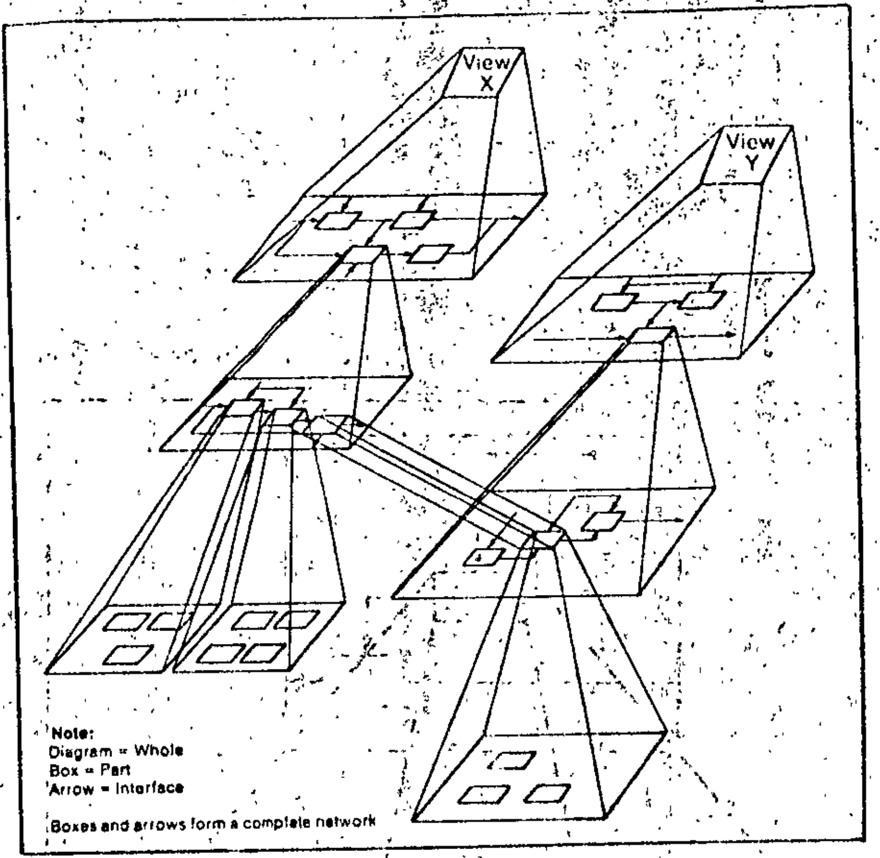
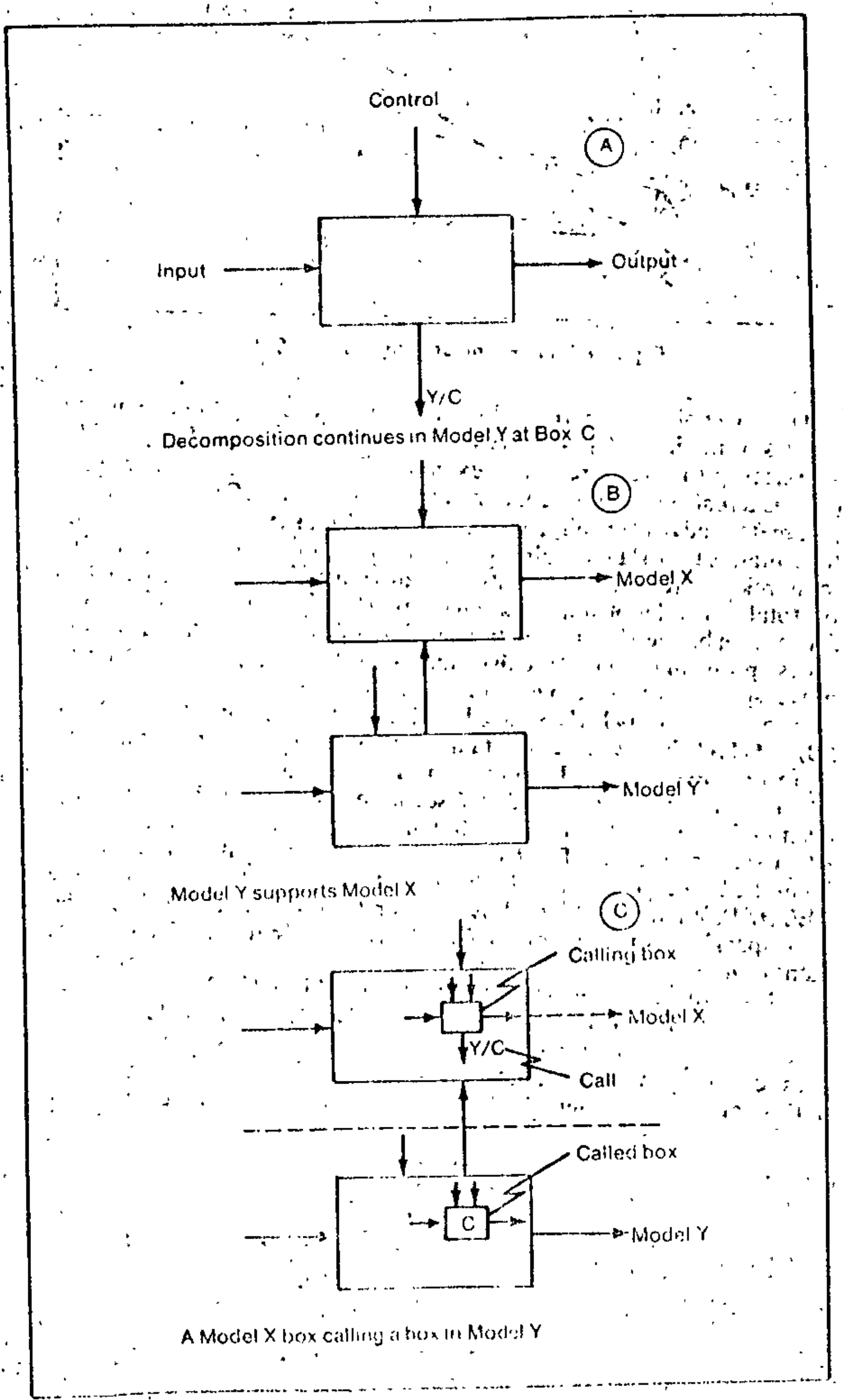


Figure 2. Two Interconnected Models of a System from Different Viewpoints

decisions can be encapsulated within a mechanism for greater modifiability.

The SADT call notation corresponds very closely to the subroutine call concept of programming languages. In Figure 3A, the notation shown on the diagram of Model X indicates that decomposition from the Model X viewpoint is continued in Model Y from the Model Y viewpoint at the box identifed as C. Model Y is said to support Model X, and this is shown graphically in Figure 3B. Figure 3C shows a box in Model X calling a box in Model Y, which corresponds to the situation shown schematically in Figure 2.

The call reference expression may, in fact, be conditional, giving several different called boxes, each one flagged with a condition specifying the particular box to be called to supply detailing. Similarly, a mechanism may support many different models, and from any one of those models an ordinary or a conditional call can be made. Thus, a node can be shared among many models (as is represented in Figure 4). These provisions allow precise representation within a single overall design of alternative design approaches. In addition, commonalities of any sort can be modeled once and applied in many places.



Decomposition continues in Model Y at Box C

Model Y supports Model X

A Model X box calling a box in Model Y

Figure 3.[illegible] [illegible] Call Notation

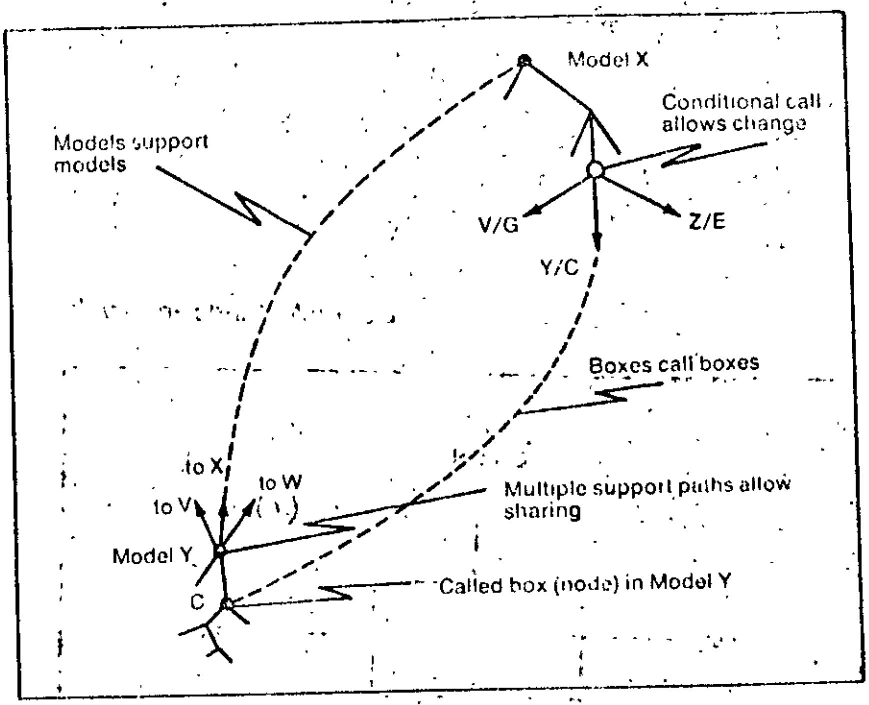

Model X
Conditional call allows change
Models support models
V/G
Z/E
Y/C
Boxes call boxes
to X
to W
to V
Multiple support paths allow sharing
Model Y
C
Called box (node) in Model Y

Figure 4. Change and Sharing Through Calls

The example in Figure 5 illustrates the use of a mechanism that models a monitor [4] — a system-structuring concept that provides safe access to a resource. The example, drawn from Brinch Hansen [1], is a real-time pair of concurrent processes. RECORD measures a parameter and periodically, as controlled by Clock A, updates a cumulative total. OUTPUT periodically, as controlled by Clock B, reads the total, resets it to zero, and prints the sum, which is computed based on the total. The relationship between the two functions is shown in Figure 5A. As can be seen by the splitting and joining arrow TOTAL, a shared-access problem to TOTAL exists and, if not handled properly, the two processes will not work together accurately.

A monitor for TOTAL is introduced to solve the problem (see Figure 5B). TOTAL is now internal data to the monitor and, therefore, the representation of TOTAL is known only to the monitor.

The monitor is represented by a separate SADT model; the top-level diagram in the monitor model is shown in Figure 5C. Decomposition of RECORD and OUTPUT, which make calls to the monitor, is shown in Figures 5D and 5E. Figure 5D shows that the TALLY operation of the RECORD function is realized by the monitor operation UPDATE TOTAL specified by the downward CALL arrow. To perform TALLY, the monitor must:

1. Activate GET TOTAL to supply TOTAL to UPDATE TOTAL.
2. Activate UPDATE TOTAL to yield the new total value.
3. Activate STORE NEW TOTAL to store the updated total.

In reviewing the figures, note that some arrow labels on Figure 5C do not match those on Figures 5D and 5E. This is allowed since SADT

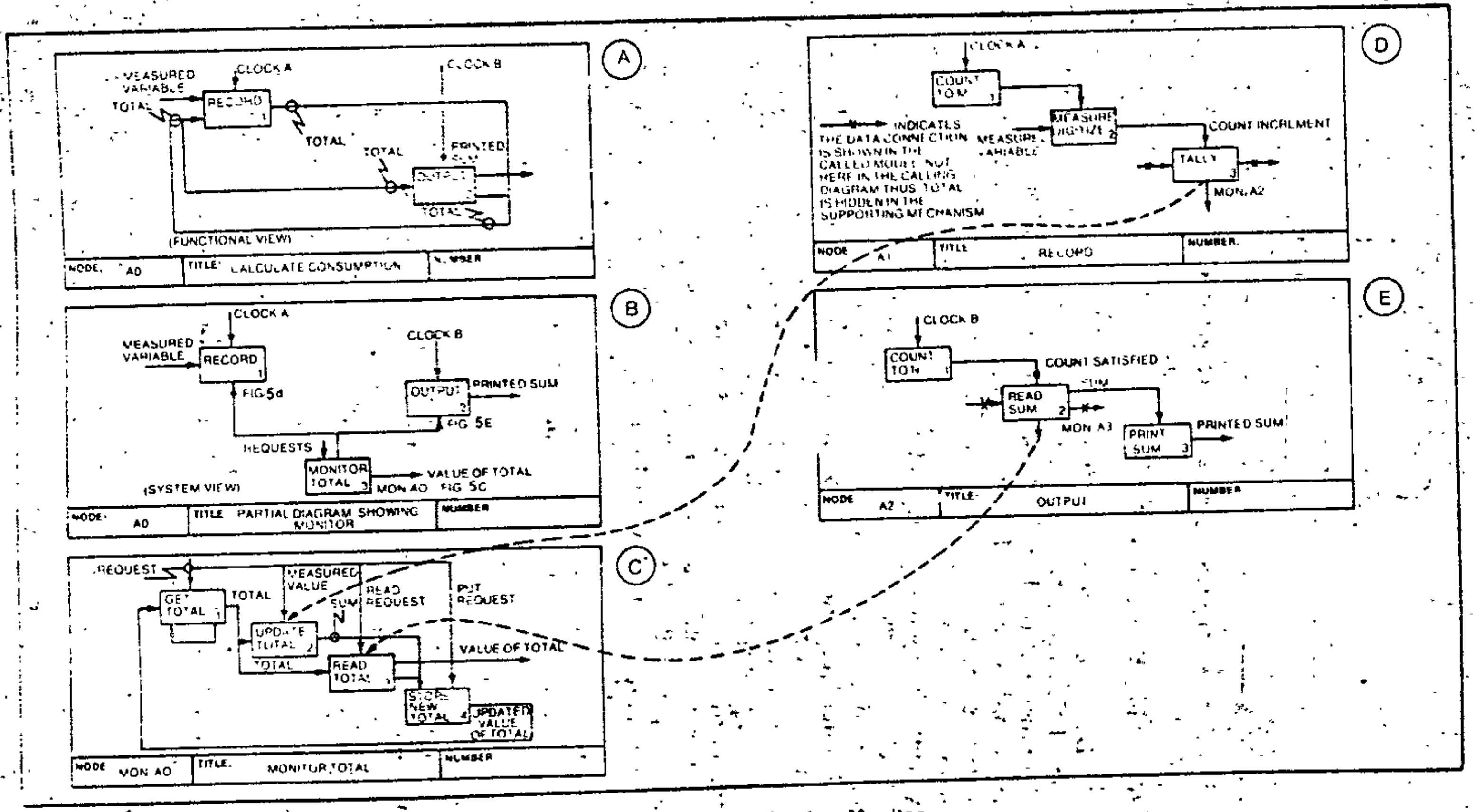
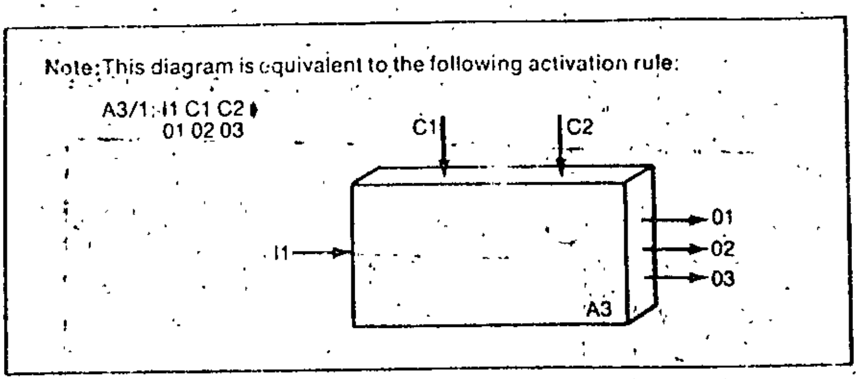


Figure 5. Modeling a Mechanism Monitor

interface-matching is based on ICOM (input-control-output-mechanisms) codes (see, for example, arrows C1, C2, O1, I1, etc. in Figure 6): not on spelling. Similarly, the activity names in Figure 5C do not match those of the calling activities in Figures 5D and 5E. The SADT call reference expression written beside the call arrow (⇜ MON/A2) uniquely identifies the called activity in the mechanism model.



Note: This diagram is equivalent to the following activation rule:

A3/1: I1 C1 C2 ➧ 01 02 03

Figure 6. Activity Box and Activation Rule

Traceability Between Models

SADT models representing different viewpoints are not independent; their relationship must be demonstrated to be consistent. The SADT tying process is used to document the relationships between models, that is, to show that all of the functions shown in a functional model are carried out by the design model. The tying process uses the SADT reference language [12] and concept of mechanisms in order to show explicitly the interconnection between models.

Each design model is built from the previous functional model and must be tied back to its predecessor to ensure the correctness of the design step [11]. Requirements traceability follows immediately, since requirements definition initiates the process. In other words, requirements definition produces a functional model depicting precisely what the system must do, as well as such system design constraints as performance criteria; the need for robustness and recovery, target machine(s), and implementation language(s). Analysis determines what functions must be performed; design structures these functions for optimal understandability, modifiability, and maintainability; specification then selects algorithms to realize the key functions. Hence, an algorithm in a specification model ties back (realizes, implements) into a design module that, in turn, ties to a function in the requirements model.

Sequencing

Arrows in Structured Analysis represent constraining relationships, not control flow. Nevertheless, because a box at the beginning of an arrow must in some sense precede the box at the end of the arrow, these precedence relations (not necessarily sequential, but implying many potential parallelism(s) do impose a kind of sequencing on the diagrams of a model. SADT diagrams represent all the parallelism implicit in the design until the designer decides to impose explicit sequencing constraints.

Activation Rules. The SADT notation for activation rules provides a way of communicating both information that would be shown by more detailing of the model and sequencing constraints that the designer wishes to add to the model. Activation rules employ logic and finite state machine notation, superimposed on the diagrams, in order to indicate in human-readable form the sequencing information known to a designer. They are of the general form.

< box-i.d. >/< rule-# >:< pre-conditions > ♦ < post-conditions>

where < *box-i.d.* > is the name of the box to which the rule applies, < *rule-#* > is a sequential numbering of the rules, <*pre-conditions* > is the set of conditions necessary for this activation of the box, and < *post-conditions* > is the set of conditions produced by this activation. Activation rules can be used to provide information to the reader who would otherwise assume the standard rule for interpreting an SADT activity box. The standard rule states that an activity box is constrained from producing its outputs by the absence of the contents of any one of its input or control arrows; only when all are present does it produce its outputs, and then it produces all of its outputs.

The situation often arises that some subset of the inputs and controls is sufficient to produce some subset of the outputs, for example, in Figure 6 if the presence of I1 and C1 were sufficient to produce 02, and I1 and C2 were sufficient to produce 01 and 03. This is information about the contents of A3 and therefore belongs in the detailing of it (see Figure 7).

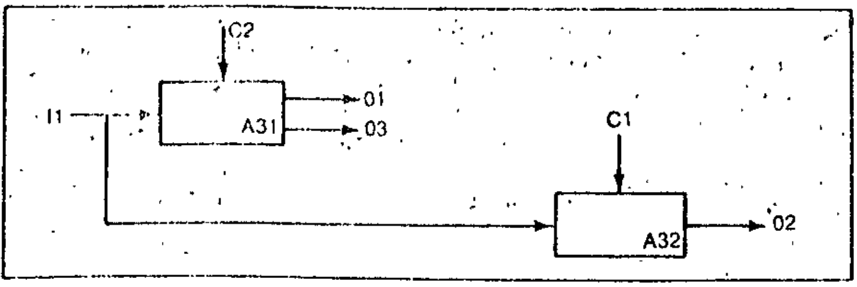


Figure 7. Detailing of Activity Box A3

In the absence of this detailing, the following activation rules can be used to relate the outputs to the inputs and controls:

3/1 : I1 C1 ♦ 02
3/2 : I1 C2 ♦ 01 03

Note that the pre-conditions list for a rule may specify the absence of an input or control rather than its presence. Such negation is indicated by a bar over the appropriate pre- or post-condition.

$3/1 : I1\ \overline{C}1\ \overline{C}2$ ♦ 02
$3/2 : I1\ \overline{C}1\ C2$ ♦ 01 03

Activation of a box can result in the cancellation of one of the pre-conditions; that is, part of the post-condition is that one of the pre-conditions has been negated. This is indicated by putting the appropriate term in the post-conditions with a negation bar:

$3/2 : I1\ \overline{C}1\ C2$ ♦ 01 03 I1

In such case, we say that I1 has been consumed. Figure 8 illustrates the consumption of an input and a post-condition indicating that one or the other of two outputs is produced.

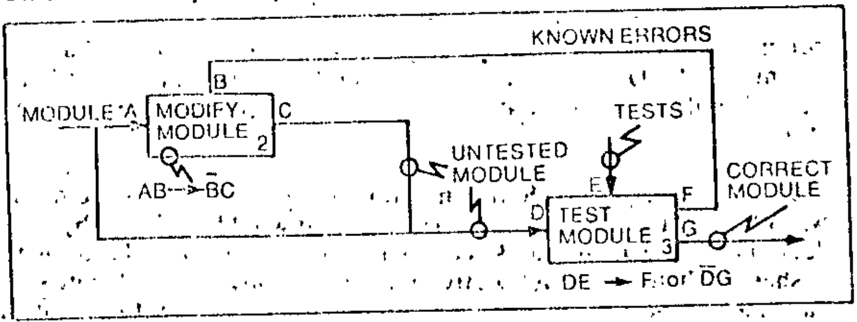


Figure 8. Consumption of an Input

The detailed design diagram shown in Figure 9 is an example of an SADT diagram containing potential parallelism. The function CREATE ENTRY could be performed in parallel with DETERMINE PRIORITY. Although the notation makes the potential parallelism explicit, the designer can easily impose a sequence on the diagram. For example, to indicate that DETERMINE PRIORITY must be performed before CREATE ENTRY, the following two activation rules are used:

4/1 : DK ♦ SM
2/1 : SFG ♦ $\overline{S}$H

In this rule, S is shown as an artificial output of DETERMINE PRIORITY that indicates that the function has been performed. S is then consumed by CREATE ENTRY. The artificial output, which was created only to impose sequence, is shown as a dotted line on the diagram.

Activation rules must be consistent throughout a model, and it is possible to verify that all the activation rules on a diagram correspond to the activation rules of the parent box.

Detailed design diagrams, annotated with activation rules and quantitative information about the characteristics of data and activities, provide the basis for producing module specifications from which code can be generated. A variety of specification methods have been used with SADT to bridge the detailed SADT design diagrams to the code. Generally, module specifications are used when the details of the processing steps are the most important thing remaining to be specified. The SADT activity model shows the design structure, the interfaces between components, and the sequence in which components would be executed. The

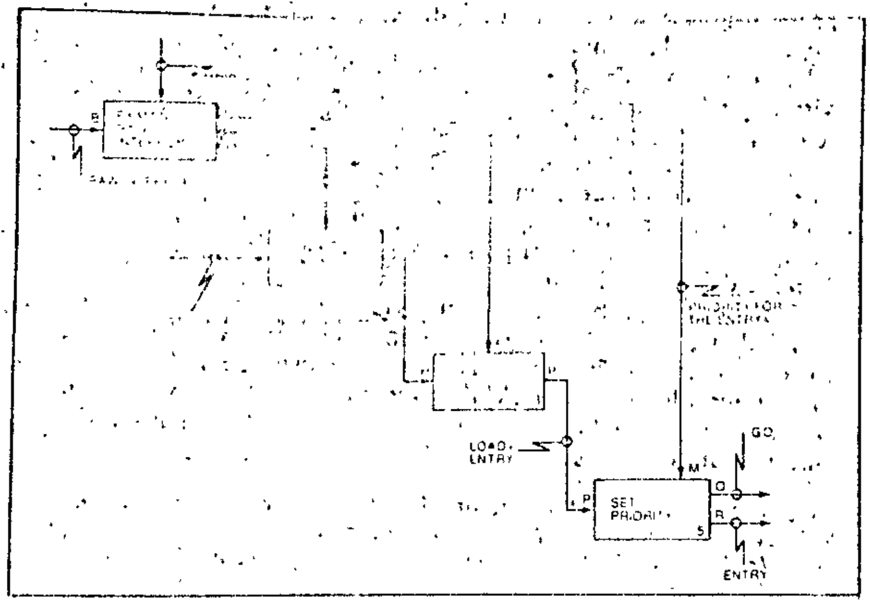


Figure 9. Detailed Design Diagram with Sequence Specified

module specifications would normally show the detailed processing logic and the data definitions that would be used in writing the program

RELATIONSHIP OF SADT TO CONTEMPORARY DESIGN METHODOLOGIES

A design methodology should specify:

- How to recognize a good design
- How to create a good design
- How to communicate a good design

Here, good depends on a method's purpose or goals that, in turn, largely determine how "good" modules will be selected and connected. The following definition for modularity has been given [15]:

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

We might characterize the purposes of three widely publicized approaches as:

- Structured Design [17] — Emphasize functional decomposition.
- Jackson Approach (see portfolio 35-05-02) — Program structure should mirror the problem structure.
- Parnas Approach [9] — Hide design decisions (e.g., data structure representations).

All three methods are compatible with using SADT as the vehicle for communicating design. Each of the three design methodologies [7] includes its own design documentation concepts, but they are not integral to the aspects of the design methodology concerned with recognizing and creating a good design. This makes it possible to exploit the design guidelines of each methodology within the common framework of SADT.

Structured Design concepts, for example, have proven to be useful in guiding SADT authors in the choice among alternative decompositions [3]. In particular, the concept of coupling has been used to judge whether the arrows that connect diagrams are unacceptably complex. The concept of cohesion has helped authors judge whether activities are related closely enough to belong on one diagram.

The Jackson methodology documents the structure of input and output data with a graphic notation that represents a part of the information contained in SADT data models. SADT also allows natural extension of the Jackson method to those cases in which design structure should not follow the given data structure exactly.

The SADT mechanism notation also provides a concrete way of representing Parnas' information-hiding modules, as well as such abstractions as virtual machines, monitors, abstract data types, and other useful design constructs. It also provides ready expression and control over virtually all of the aspects of software engineering that have been found to be of practical significance [15].

In summary, SADT does not force the designer to accept a new, or even any particular, design methodology. Rather, it disciplines and directs a designer's thinking while enabling one directly to draw, see, compare, and apply design alternatives in any rational way that suits the specific design conditions.

Finally, and perhaps most important, the rigor of SADT, which is enabled by the graphic language syntax and semantics and enforced by the disciplined routine of the reader/author critique and correction cycle, allows full progress tracking throughout the design and implementation process. Visibility, understandability, and confirmation of details throughout the process enable effective project management.

This portfolio is an edited version of an article, written by D.T. Ross, M.E. Dickover, and C McGowan, of SofTech Inc, that appeared in *Proceedings of ACM '77*. © 1977, Association for Computing Machinery Inc. Reprinted with permission.

References

1 Hansen, P Brinch *Operating System Principles* Englewood Cliffs NJ Prentice-Hall, 1973
2. Connor, Michael F. "SADT for Requirements Analysis and Specification" *Systems Development Management AUERBACH Information Management Series*, Portfolio 32-04-02 1980
3. Dickover, M E. "Principles of Coupling and Cohesion for Use in the Practice of SADT." Tech Publ. No 039 Waltham MA: SofTech Inc, 1976.
4 Hoare, C.A.R. "Monitors: An Operating System Structuring Concept." *CACM*, Vol 17, No. 10 (October 1974), 549-557.
5 Hori, S "Human Directed Activity Cell Model" CAM-I Long Range Planning Final Report. CAM-I Inc, 1972
6 Jackson, M.A. *Principles of Program Design* London Academic Press, 1975
7. McGowan, C L., and Kelly, J.R. "A Review of Decomposition and Design Methodologies" *Proceedings of Infotech Conference on Structured Design*, 1977
8. Melich, M. "Evolution of Naval Missions and the Naval Tactical Data System" *Proceedings of the Conference on Managing the Development of Weapon System Software*, 1976.
9 Parnas, D L. "On the Criteria to be Used in Decomposing Systems into Modules" *CACM*, Vol 15, No 12 (December 1972), 1053-1058.
10 Payne, R. "Calling for Attention...ITT's Interesting Telephone System." *Computing Europe* (June 1977).
11. Ross, D.T "Quality Starts with Requirements Definition" Position Paper for IFIP TC2 Working Conference on Constructing Quality Software Novosibirsk USSR, May 1977.
12. Ross, D.T. "Structured Analysis, A Language for Communicating Ideas." *IEEE Transactions on Software Engineering*, Vol. 3, No 1 (1977), 16-34.
13 Ross, D T., and Brackett, J W "An Approach to Structured Analysis" *Computer Decisions*, Vol. 8, No 9 (1976) 40-44.
14 Ross, D.T., and Schoman, K E "Structured Analysis for Requirements Definition" *IEEE Transactions on Software Engineering*, Vol. 3, No. 1 (1977), 6-15.
15. Ross, D T., Goodenough, J.B., and Irvine, C.A. "Software Engineering Process, Principles, and Goals." *Computer*, Vol 8, No 5 (May 1975), 17-27.
16 Thomas, M "Function Decomposition SADT." *Infotech Structured Analysis and Design*, 1978, 335f.
17. Yourdon, E., and Constantine, L L. *Structured Design* New York: Yourdon Inc, 1975.
18. Zimmerman, M. "ICAM, Revolution in Manfacturing" *Machine Design*, Vol. 49, No 1 (1977)

holm, Sweden, 1979.

[19] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming*. Reading, MA: Addison-Wesley, 1979.

[20] B. Liskov and V. Berzins, "An appraisal on program specifications," M.I.T. Tech. Rep., 1977.

[21] H. M. Markowitz, "SIMSCRIPT," RC 6811, IBM Res. Div., 1978.

[22] T. W. Mao and R. T. Yeh, "Communication Port: A language concept for concurrent programming," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 194-204, 1980.

[23] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128-138, 1970.

[24] F. Parr, "Software maintenance," in *Proc. Beyond Structured Programming*, INFOTECH State of the Art Conf., 1978.

[25] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223-252, 1977.

[26] W. E. Riddle and R. E. Fairley, Eds., in Proc. Software Development Tools Workshop Conference (Pingree Park, CO), May 1979.

[27] L. Robinson and R. C. Holt, "Formal specifications for solutions to synchronization problems," Comput. Sci. Group, Stanford Res. Inst., Stanford, CA, 1975.

[28] D. T. Ross, "Structured analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16-34, 1977.

[29] N. Roussopoulos, "A semantic network model of data bases," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Toronto, Toronto, Ont., Canada, TR 104, 1976.

[30] H. Simon, *The Sciences of the Artificial*. Cambridge, MA: M.I
Press, 1970.

[31] J. Smith and D. Smith, "Data base abstractions: Aggregation
generalization," *Assoc. Comput. Mach. TODS*, vol. 2, pp. 1
133, 1977.

[32] D. Teichroew and E. H. Hershey, "PSL/PSA: A computer-ai
technique for structured documentation and analysis of infor
tion processing systems," *IEEE Trans. Software Eng.*, vol. SE
pp. 41-48, 1977.

[33] R. T. Yeh, *Current Trends in Programming Methodology, Vo
Software Specification and Design*. Englewood Cliffs, NJ: P
tice-Hall, 1977.

[34] R. T. Yeh, A. Augustine, R. Mittermeir, W. Mao, and F. Ev
"Software requirement engineering: A perspective," in *P
Beyond Structured Programming*, INFOTECH State of the
Conference, London, England, Nov. 1978.

[35] R. T. Yeh, N. Roussopoulos, and P. Chang, "Systematic der
tion of software requirements through structured analysis," T
Rep. SDBEG-15, Dep. Comput. Sci., Univ. of Texas, Au
Sept. 1979.

[36] P. Zave, "A comprehensive approach to requirements problem
in *Proc. COMPSAC '79* (Chicago, IL), pp. 117-122, Nov. 197

[37] P. Zave, "Formal specification of complete and consistent per
mance requirements," in *Proc. 8th Texas Conf. Compu
Systems* (Dallas, TX), pp. 4B-18-4B-25, Nov. 1979.

Software Representation and Composition Techniques

LAWRENCE J. PETERS

Abstract—The field of software development has undergone some of its most profound changes in the last ten years. Much of this change has been in response to ever increasing demands on software systems in terms of their complexity, reliability, and resiliency. Symptomatic of such rapid evolution is the proliferation of methods and techniques intended to solve "the" software problem. However, real-world software design problems often exhibit characteristics that make them unique. This forces the software engineer to seek alternative ways of composing and documenting a design. This paper describes a sampling of the alternatives and directs the reader to sources of more detailed information regarding their use as well as a larger survey of this rapidly changing field.

I. Introduction

SOFTWARE development challenges the software engineer in several ways. Unlike many other fields, systems of software will not be mass produced. This divorces the software engineer from many problems associated with ma
facturing. However, since he is dealing with logic—the abstr
—the results of his labor are difficult to identify with. So
ware operates on a time scale and reference frame which is
comprehensible by human standards. These factors, and
lack (for the most part) of an engineering background on
part of software developers, have led to the naive view th
software design is unique and that its problems are exclusiv
those of software. The software engineer also has the le
guidance of any technical field regarding the scope of his pr
lem or the acceptability of his solution. For example, in
craft design, a set of strictly adhered to parameters calle
design envelope are established early in the design effort.
task of the aircraft design team is to create a design wh
meets the requirements and falls within the envelope. Fact
included in the envelope may include size, weight (dry
maximum takeoff), range, and other parameters dictated
marketing and manufacturing considerations. In such hardw
oriented efforts, design envelopes are the result of experien
common practice, economics, and production capabilities.
the case of software, a concept such as the design envelop

Manuscript received March 18, 1980; revised May 23, 1980.

Portions of this article are excerpted from [11] and appear here with the permission of the publisher.

The author is with Yourdon, Inc., 1133 Avenue of the Americas, New York, NY 10036.

0018-9219/80/0900-1085$00.75 © 1980 IEEE

less straightforward—a large experience base related to a particular type of software is not available, economics is a matter of "black magic," and there is no common practice or discipline. This is particularly true in the case of software design documentation. Hence, the software designer has little or no guidance in the execution of his task(s).

Several authors have attempted to rescue the software engineer from this dilemma by publishing a wide variety of tactics and strategies with which to address software development. Each author relates the efficacy of his approach to their own experience usually in some limited area of application. Each utilizes a system of representation techniques designed to transmit the attributes of the design considered important by such authors. We will examine representative examples of these, comment on their source and utility, and finally describe current trends and what they signal for future technological developments.

II. Background

The emphasis in early software development was on obtaining a program which worked. That is, it gave answers which agreed with accepted values or, where accepted values did not exist, saved large amounts of manual labor. For example, Gustav Mie [1] solved Maxwell's equations for the case of electromagnetic waves passing through a colloidal suspension containing dielectric spheres in about 1905. In the 1930's, using comptometers and a lot of manual labor, less than a few dozen of these coefficients useful in estimating information loss in infrared, microwave, and laser transmission had been computed. By the mid-1960's, several thousand values could be computed in a matter of seconds [2]. But by the 1960's, other more challenging problems were being attempted. Instead of developing single programs or small sets of programs, large assemblages of programs were being attempted. This ushered out the age of functional programming and ushered in the age of structure oriented programming. The problems created by attempting to control satellites or air traffic with these large systems highlighted the need for some sort of philosophical viewpoint which would enable software designers to create systems (not just single programs) that worked; systems whose construction was aided by the design and not encumbered by it. During this time many concepts were "discovered" in software such as the difference between logical design (i.e., abstract, conceptual) and physical design (i.e., blueprint, one-for-one correspondence to what will be built), philosophies of design (e.g., top-down), and the use of prototypes from which to learn and protect an investment. Much of this ferment peaked in the late 1960's and early 1970's with the advent of some specific methods and approaches to the problem of software design representations. These seemed to address structure problems and some were adopted quite widely. However, they have been found to be in need of some support in order to solve "real world" problems. This has resulted in yet another wave of approaches. This body of help now available to the software engineer falls into two classes—methods (strategies, recommendations, or guidelines based on a philosophical view) and techniques (tactics or well-advised "tricks of the trade"). Both classes are described in the remainder of this paper together with a look at where much of this may be leading.

III. Representing Software Designs

One of the earliest techniques to be developed is that of software design representation. The problem is a fundamental one. That is, how does the software designer depict his design s as to communicate it to fellow workers and the customer Architects can use sketches and the customer's everyday ex periences with physical surroundings to capture the overa characteristics of a proposed building. But software is not directly experienced product.

There are several issues related to the portrayal of a soft ware design [3]. These can be organized into the followin categories.

1) Architectural—The depiction of the relationships whic exist between major system elements and between the softwar system and the outside world (user, terminals, etc.).

2) Structural—The depiction of relationships between dis tinguishable system elements. This is a static view of th system.

3) Behavioral—The depiction of interactions among syster elements. This is a dynamic view of the system.

4) Informational—The conceptual organization of the dat used by the system.

The sensitivity of project success to these issues has increase as the complexity of the software system being attempted ha increased. The role of software representation techniques ha shifted from documenting what the design contains to tha of playing an active part in the evolution of the design. Th evolution is the result of increased communication among th designer, user, customer, and program implementation pe sonnel. Some design representation techniques can be used t describe system level concepts while others are at a more loc level, often closely tied to a specific subset of issues. Howeve all are directed at reducing and controlling the amount c complexity exhibited by these abstract models of the systen We will examine some of the approaches that are available fo architectural, structural, and behavioral classes of informatior

A. Representing Architectural Features

Only recently has this important aspect of software desig representation been addressed in the literature [4]. Perhap this is because it is easier to address local, simple, and compre hensible aspects of an overall system than to deal with thes "global" issues. The prospect can be overwhelming.

The Leighton diagram is intended to address the proble of depicting software system architecture in an easy, to rea and understandable format. The approach taken tends t avoid many of the problems associated with employing hie archy diagrams while displaying the sources of inputs, pr cessing levels, precedence relationships, and destinatio of outputs.

Leighton diagrams were originally developed to serve a fün tion analogous to that of the artist's concept in engineerin This diagrammatic form is intended to satisfy its author's pe ception of the need to be fulfilled

1) employ some of the characteristics of treelike structure
2) clearly depict sources of input and destinations of outpu
3) be easy to present and copy using $8\frac{1}{2} \times 11$ in. for and allow sequential presentation to avoid repeti backtracking;
4) clearly depict the complexity associated with what being presented and the occurrence/use of comm routines;
5) permit the incorporation of textual information;
6) be effective on a broad spectrum of system types;
7) be cost effectively automatable.

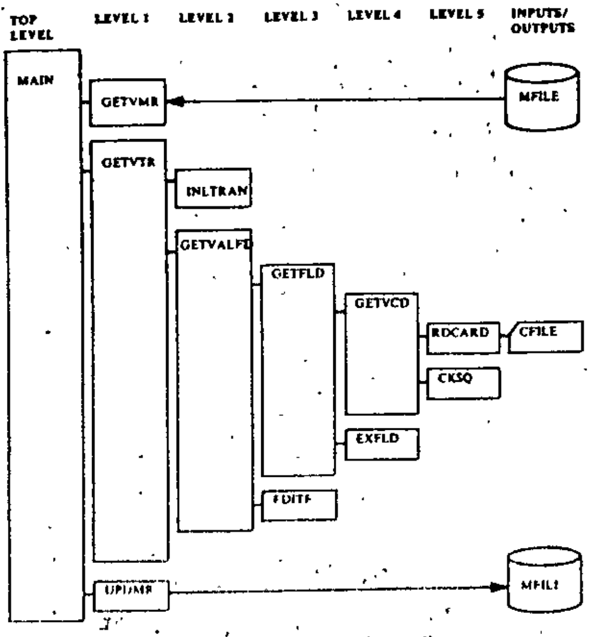


Fig. 1. An example of a Leighton diagram.

The primary goal in developing this diagrammatic technique was to enhance the communication between customers, users, software architects, and software development personnel concerning design (and development) issues.

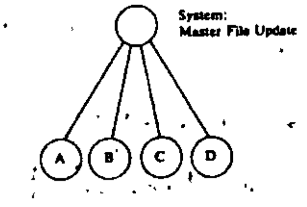
This first of many different software design representation schemas for depicting system architecture is presented in Fig. 1. Note how the essential elements (e.g., external interfaces, interactions among major software components, and overall priority or precedence relationships) are all that is depicted. The flavor of what the system is like is retained without the introduction of unnecessary detail. In the sense that software design representation began with descriptions of individual programs and only recently turned to describing overall systems, this process has been bottom up in nature.

B. Depicting Structural Characteristics

Portraying the structure of a software system involves the taking of a "snapshot" of the relationships between internal system elements. It is a static view of what the system is like. We will first look at an approach which was common to engineering at least a couple of thousand years before it was "discovered" as part of top-down structured programming approaches. The other representation schemes (methods) in the remainder of this article utilize the design tree concept together with some other features.

1) The Design Tree: This approach [5] to depicting design structure is directed at externalizing the composition of a system. It is based on concepts in mathematics which have been used in top-down design.

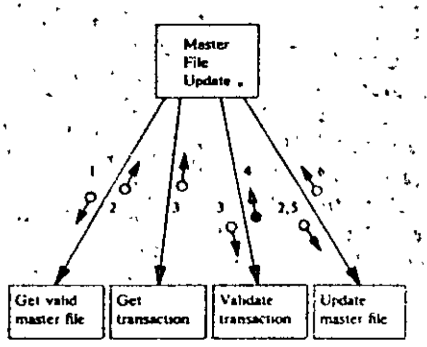
Those utilizing the design tree concept employ one of two mutually exclusive concepts. The first, containment, refers to the fact that a concept includes (or contains) other concepts. For example, the concept of airplane includes the concepts of propulsion system, body, wing, and tail. The second, hierarchy, explicitly portrays concepts as being subordinate to one another. That is, not just including other lesser concepts, but being greater or higher on a scale in some way. This means of ordering systems has been used since the time of the Egyptians (and possibly earlier). It is inherent to the concept of top-down design and analysis. Thus the de tree supports top-down decomposition both conceptu and graphically (Fig. 2).



PROCESSES

A Get valid master file
B Get transaction
C. Validate transaction
D. Update master file

Fig. 2. An example of a design tree.



NOTES

1. Customer master file
2. Valid master file
3 Transaction
4 Transaction "OK" flag
5 Validated transaction(s)
6 Updated master file

Fig. 3. A structure chart.

The design tree uses a simple flexible notational scheme is simple in that only two basic graphics elements are used straight line and a node symbol. Examples of commonly u node symbols include circles, squares, and rectangles. Each these will contain textual information. The user of this sche can employ any node symbol that is convenient or may cho not to enclose the text associated with each node. This sche is also flexible in that it can be used to depict complex simple systems to the desired level of decomposition.

2) Structure Charts: Structure charts were originally de oped by Constantine *et al.* [6] to specify modular characte tics of software during design. They are an integral part of structured design method [7], [8].

The basis for the structure chart is a treelike structure wh depicts hierarchical relationships. However, the basic notat of the design tree has been enhanced to externalize module lationships. A module is defined as a set of lexically contigu program statements which can be referred to by name. relationships and interactions which are depicted include flow, activation, and communication of control paramet Note that the design tree can relate concepts or processes comprise the system. This scheme specifically identifies m ules that will comprise the software system.

Structure charts can be drawn in several different w The approach proposed by Constantine utilizes three b graphical forms

1) the rectangle, used to contain a module or module descriptor;
2) the vector, used to highlight interaction between modules (usually a call);
3) the arrow with circular tail, used to depict the transfer of data and control between modules.

Together, these comprise the structure chart as shown in Fig. 3. In this case, the system level is a master file update processor which has several subordinate functions. The master file is passed to the edit module which returns errors and valid entries and so on. Note that control elements are identified by having the circles on their tails filled in.

3) Structured Analysis and Design Technique: This approach is usually referred to as SADT$^{\text{TM}}$. It was originated and trademarked by SofTech Inc. [9]. It was derived from work done by Hori [10], and as a result of experience with computer-aided manufacturing studies.

This representation scheme utilizes two forms. One is a tree-like structure (a design tree) which acts as a roadmap to the system model. The other form is the activity chart. The system model can be used to describe a single program or group of programs. It is composed of one or more activity charts or more simply diagrams. The basic idea here is to provide the user of the technique with a means of graphically portraying whatever his analysis of an existing system reveals or his perception of a system under design.

This method also uses labeled rectangular boxes and arrows. Several distinctions are made both in what is represented and how it is represented. For example, the basic distinction between data flow and activities is made but data flows are classified as being input, output, or control (Fig. 4). However, execution sequences are not explicitly shown.

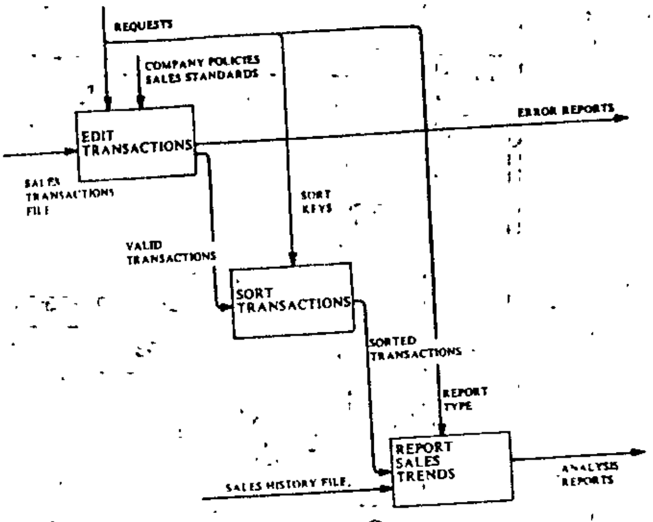


Fig. 4. An SADT$^{\text{TM}}$ diagram.

C. Representing Behavioral Features

The issues related to the representation of the actual behavior or "execution" of the software are among those first addressed in the early days of software design. These involved the detailed and explicit documentation of precisely what the (eventual) program would do when it was constructed. One of the first of these schemes, the flowchart, typified the attitudes and concerns of the times. Namely, will the program work and if so, how?

Today, attitudes have changed. The concern is focused more on the organizational character of the program than on the details of execution. Problems with program maintenance and development have emphasized the need for some reordered priorities. The approaches presented here are a representative sampling of a much larger set of schemes from which to choose [11].

1) The Flowchart: The flowchart is probably the most widely used, misused, and misunderstood software representation schemes in use today. It has resulted in many derivatives. It was originally developed by von Neumann who intended it to be an accurate means of documenting a program after it was written not as a design representation scheme. The use of the flowchart has been somewhat standardized through the efforts of the American National Standards Institute (ANSI) [12]. What is presented here employs the ANSI format.

The ANSI flowchart depicts the details of control flow. This is accomplished by using different symbols for processes, decisions, and entry and termination points. There are no specific requirements that some specific set of constructs be employed. Hence, any type of control transfer supported by a programming language or algorithm may be represented via the flowchart.

Primarily this scheme uses three symbols: the rectangle, the rhombus (or diamond), and a directed line. Other symbols are used to depict manual operations and external/hardware interfaces. The rectangle is used to represent arithmetic operations or processes. Examples of these include "$n = n + 1$," "$x = ab$," and reset end of file flag, etc. The diamond shape is used to represent decision points. Examples of these include end of file flag set, "$n = 0$," and "$x - ab$." The directed line indicates the flow of control or precedence in a flow sequence. An example of an ANSI flowchart is presented in Fig. 5.

2) Decision Tables: The decision table has been used to analyze and describe deterministic systems and to sort out confusing decision making problems. Their use in programming is not new and dates back to the days of wired logic and telephone switching problems and beyond [12].

Decision tables can take many different forms [13]. These all stem from the same basic notion that for each of the possible combinations of situations that a system (or program can encounter, the system's response is known. These situations are referred to as conditions while system responses ar referred to as actions. For every condition or set of condition which can occur, one and only one action or set of actions ca occur. The response of the system is known with certainty.

The basic decision table consists of two portions—the cond tion stub and the action stub. Conditions are collected an (optionally) labeled into the condition stub while actio are collected and (optionally) labeled in the action stub. Co ditions and actions are most often described horizontall Vertical columns in the condition stub are used to identi which conditions apply in a given instance: A correspondi column in the action stub describes the system's respo (Fig. 6).

3) Hamilton and Zeldin Approach: This scheme was o inally devised in support of the National Aeronautic Space Administration (NASA) space shuttle software devel ment [14]. It is a response to the need to simplifying clarifying the often overwhelming detail present in flowcha

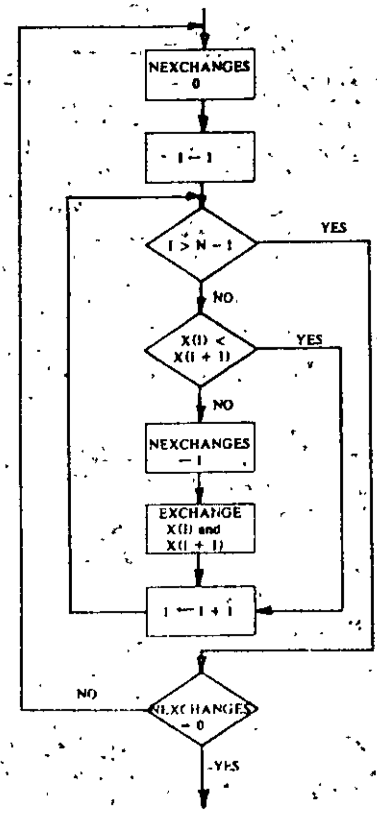


Fig. 5. An ANSI flowchart.

		1	2	3	4	5
A	Start	Y	N	N	N	N
B	Number of exchanges = 0	–	–	N	Y	–
C	Array index within limits	–	Y	N	N	Y
D	Indexed pair out of order	–	Y	–	–	N
E	Reset start flag	X	–	–	–	–
F	Initialize number of exchanges = 0 and set array index = 1	X	–	X	–	–
G	Exchange array elements and set exchange flag	–	X	–	–	–
H	Increment array index	–	X	–	–	X
I	Exit	–	–	–	X	–

Fig. 6. A decision table representation of an algorithm.

The Hamilton and Zeldin design representation scheme has been further refined and enhanced and is now supported by an automated flowcharting tool [15]. Diagrams of this type are called structured design diagrams (not to be confused with the "structured design method," Section IV).

The original intent of this scheme was to support the use of higher level languages. However, it could be employed in other applications. The objectives of this scheme include

1) explicitly depicting the levels of nesting (and hence, complexity) inherent in the design structure;
2) depicting all processes in an algorithmic manner (as opposed to employing textual material);
3) demonstrating the extent or scope of control/effect of all loops;
4) supporting and encouraging the use of the basic constructs and their alternative forms.

The basic effect of this scheme is that the software designer and customer can follow sequences of equations to determine whether or not they are appropriate while gaining an appreciation for the relationship(s) between different parts of a software system.

Structured design diagrams are composed of rectangles, directed lines, and a pentagonal combination of a rectangle and triangle. Execution is generally from top to bottom on the diagram with the exception of IF tests and DO loops which proceed from left to right with each occurrence. This clearly shows the level of nesting. Each of the graphic forms contains an algebraic statement or test, as appropriate. An example of this technique is presented in Fig. 7.

4) Nassi and Shneiderman Approach: This approach was introduced as a means of syntactically enforcing use of three

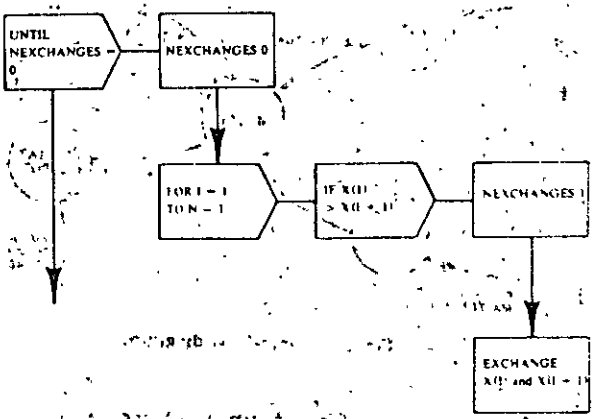


Fig. 7. The Hamilton and Zeldin design representation.

basic program constructs: sequence, decision, and loop. Si its introduction [16], it has been modified [17], [18] and been incorporated into an interactive graphics system to software design [19].

Nassi-Shneiderman diagrams are a departure from flowch based techniques. They aid in the adherence to the use of gramming structures other than the GO TO (an unconditio transfer of control). A single process is completely contai within a rectangular box. The box is subdivided into secti Each section denotes a specific subprocess (e.g., an assignm decision). A hierarchy of such diagrams can be established maintained via the use of routine calls or dummy proce which refer to other diagrams. The range of a loop, the and effect of a decision, and the general sequence of the cess flow are all explicit in this scheme.

These diagrams consist of rectangles and triangles containing a statement of the operation performed and arranged so as to define process flow (Fig. 8).

D. Comments Regarding Software Design Representation

There are well over a dozen different approaches to softw design representation. Many of these also have derivat making the total number in use today much higher [11]. documenting a software design, three issues must be addre regardless of what representation scheme is used.

1) what information should be communicated?
2) who is the audience?
3) what are the concerns of the audience?

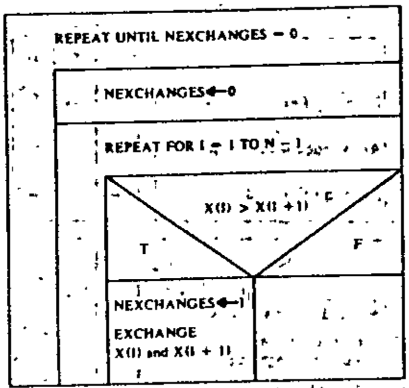

REPEAT UNTIL NEXCHANGES = 0
NEXCHANGES ← 0
REPEAT FOR I = 1 TO N − 1
X(I) > X(I + 1)
T
F
NEXCHANGES ← 1
EXCHANGE X(I) and X(I + 1)

Fig. 8. The Nassi–Shneiderman representation approach.

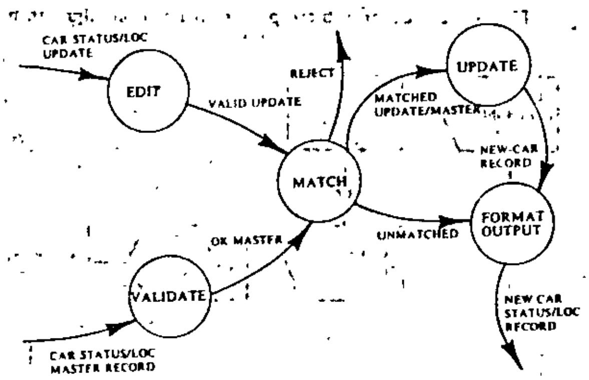


Fig. 9. A data-flow diagram.

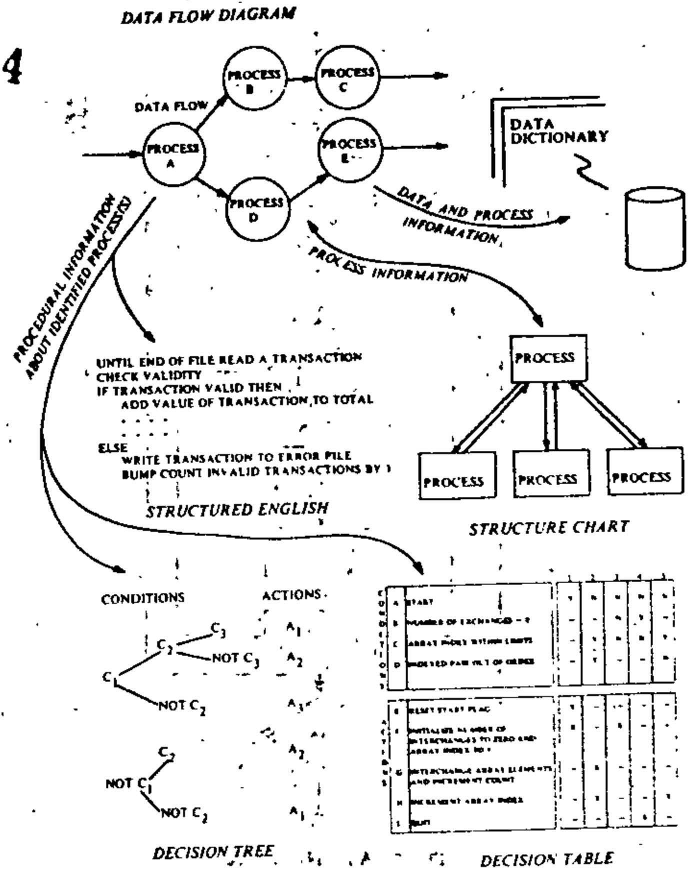

DATA FLOW DIAGRAM
DATA FLOW
PROCESS A
PROCESS B
PROCESS C
PROCESS D
PROCESS E
DATA DICTIONARY
DATA AND PROCESS INFORMATION
PROCESS INFORMATION
PROCEDURAL INFORMATION ABOUT IDENTIFIED PROCESS(S)
UNTIL END OF FILE READ A TRANSACTION
CHECK VALIDITY
IF TRANSACTION VALID THEN
ADD VALUE OF TRANSACTION TO TOTAL
ELSE
WRITE TRANSACTION TO ERROR FILE
BUMP COUNT INVALID TRANSACTIONS BY 1
STRUCTURED ENGLISH
PROCESS
PROCESS
PROCESS
PROCESS
STRUCTURE CHART
CONDITIONS
ACTIONS
C1
C2
C3
NOT C3
NOT C2
NOT C1
A1
A2
A3
DECISION TREE
DECISION TABLE
START
NUMBER OF EXCHANGES = 0
ARRAY INDEX WITHIN LIMITS
INDEXED PAIR OUT OF ORDER
RESET START FLAG
INITIALIZE NUMBER OF INTERCHANGES TO ZERO AND ARRAY INDEX TO 1
INTERCHANGE ARRAY ELEMENTS AND INCREMENT COUNT
INCREMENT ARRAY INDEX
QUIT

Fig. 10. An overview of the structured design method.

Since there are many different types of reviewers, it is unlikely that a single representation scheme will satisfy the needs of all of them. Rather, it is more likely that selection criteria and problem characterization parameters could be applied [11] in order to "engineer" a representation system composed of many different classes of representation schemes.

IV. Composition of Software

As challenging as the software design representation problem may seem, the problem of composing or creating the software in the first place is even more so. Many articles have been published which describe to the reader how software may be derived beginning with a few given pieces of information. Each method utilizes a design representation scheme or system of schemes in order to accomplish its goal(s). One approach [11] organizes these approaches into three categories

1) data-flow oriented methods
2) data structure methods
3) prescriptive methods.

The naming of these categories is related to the information they utilize as being given (e.g., a data-flow model) or the nature of the approach (e.g., prescriptive). Rather than describe the use of each of the more than one dozen major methods [11] currently documented in the literature, a representative example of each of the categories will by synopsized here.

A. Data-Flow Oriented Methods

Data-flow oriented methods provide the software engineer with guidance on how to define software given that he has a model of the data flow which will (eventually) occur in the system. A widely used approach is that of structured design [8].

This approach utilizes a network oriented design representation called a data-flow diagram and structure charts (see Section III). Together they form an integral part of a system of design representation directed at capitalizing on these depicted properties. Structured design's notational system depicts structural and behavioral characteristics separately and includes objective evaluations of design quality. These evaluations are aimed at identifying the level of system strength and flexibility (coupling), as well as the level of internal strength possessed by each model. These concepts can be applied to any software design and constitute what well may be a "natural law" of software composition. They need not be thought of as being strictly applicable to designs arrived at using the structured design method. The use of this method is complemented by the existence of a method for defining system specifications called structured analysis [20].

Structured design views systems in two complementary ways. One is the flow or movement of data while the other is the transformation(s) which such data flow undergoes to be transformed from input into output. Together they form a network model of a system. Data enter as input, undergo transformations, converge, diverge, get stored with other data, and become output. This view of software may sound simple and perhaps dull, but it generates several interesting side effects. Among these are the following.

1) Absence of time in the data-flow representation—Since movement and transformation of data are the only characteristics represented by the data-flow diagram, the concept of the

passage of time along any single or several data-flow path(s) is not present (Fig. 9). The software engineer is free to concentrate on the establishment of a clear understanding of what major/minor transformations must occur in order for the input data to be incrementally and correctly transformed into output.

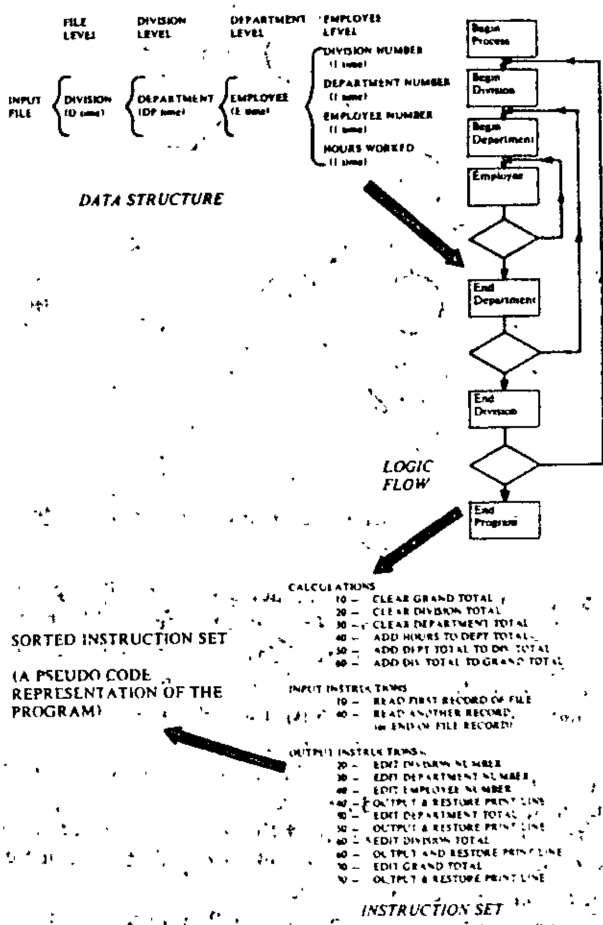
2) Lack of classical functional decomposition—Top-down design has been described as showing only one path in a tree-like structure because it assumes that there is only one problem to be solved [5]. The use of the "data-flow viewpoint" reduces the effect(s) which the designer's experience and biases will have on the results. This allows the "shape" or structure of the system to be retained. Further on in the structured design process, these "natural aggregates" of transformations and data flows play a vital role in the identification and organization of modules in the construction plan or physical structure of the system.

A serious problem for software engineers is the control of complexity. Much of this complexity is caused by concern over solving the problem conceptually while allowing implementation issues to distort this abstract view. The software engineer finds himself pulled in two directions at once. Hence, his attention is focused alternatively on high level, abstract problem issues and low level, detail implementation ones. This shuttling between two somewhat incompatible sets of issues creates an unstable environment in which mistakes will be made. Some key issue or subtle nuance regarding the problem may be overlooked. Structured design recommends a disciplined distinction between abstract or logical design and physical or packaged design. This enables the software engineer to proceed much as the architect proceeds by first establishing a conceptual solution then adjusting it to the intended operating environment (making it practical).

The basic ploy used in structured design is to identify the data flow(s) apparent in the problem and to build in both detail and structure in an iterative fashion. It is assumed that a system specification exists before design begins. This specification identifies inputs, desired outputs, and a description of the functional aspects of the system. The specification is used as a basis for the graphical depiction of the inherent data flows and data transformations. This graphical depiction is called a data-flow diagram. From the data-flow diagrams, natural aggregates of these transformations and data flows are identified. This eventually leads to the definition and depiction of the modules and their relationship to one another and various system elements called a structure chart. The system specification is reexamined, errors and omissions remedied, and the process cycled through again. Not all steps are repeated, only those deemed necessary by the software engineer. An overview of this process is shown in Fig. 10.

B. Data Structure Oriented Methods

Data structure oriented methods give the software designer a means of proceeding to a design given that the structure of the data that will be processed by the software is known. In this case, structure refers to the logical relationship(s) which exist between data elements and not to the physical format of data. The basic premise in this case is that the structure of the data and the structure of the program must be compatible. The data structured approach we will examine here is that of Warnier [21]. It shares the data structure view with an approach authored by Jackson [22]. They are widely popular in Europe



FILE LEVEL
DIVISION LEVEL
DEPARTMENT LEVEL
EMPLOYEE LEVEL
INPUT FILE
DIVISION (D time)
DEPARTMENT (DP time)
EMPLOYEE (E time)
DIVISION NUMBER (1 time)
DEPARTMENT NUMBER (1 time)
EMPLOYEE NUMBER (1 time)
HOURS WORKED (1 time)
DATA STRUCTURE
Begin Process
Begin Division
Begin Department
Employee
End Department
End Division
End Program
LOGIC FLOW
SORTED INSTRUCTION SET
(A PSEUDO CODE REPRESENTATION OF THE PROGRAM)
CALCULATIONS
10 – CLEAR GRAND TOTAL
20 – CLEAR DIVISION TOTAL
30 – CLEAR DEPARTMENT TOTAL
40 – ADD HOURS TO DEPT TOTAL
50 – ADD DEPT TOTAL TO DIV TOTAL
60 – ADD DIV TOTAL TO GRAND TOTAL
INPUT INSTRUCTIONS
10 – READ FIRST RECORD OF FILE
40 – READ ANOTHER RECORD (or END OF FILE RECORD)
OUTPUT INSTRUCTIONS
20 – EDIT DIVISION NUMBER
30 – EDIT DEPARTMENT NUMBER
40 – EDIT EMPLOYEE NUMBER
40 – OUTPUT & RESTORE PRINT LINE
50 – EDIT DEPARTMENT TOTAL
50 – OUTPUT & RESTORE PRINT LINE
60 – EDIT DIVISION TOTAL
60 – OUTPUT AND RESTORE PRINT LINE
70 – EDIT GRAND TOTAL
70 – OUTPUT & RESTORE PRINT LINE
INSTRUCTION SET

Fig. 11. An overview of the Warnier method.

and growing in popularity in the U.S. but utilize fundament different approaches to design representation.

Conceptually it is assumed that the inherent structure input and output data is

1) discernible;
2) useful as a driving force in software development;
3) insurance that a prudently structured program will res

This method employs the following four different design resentation schemes:

1) data organization diagram;
2) logical sequence diagram;
3) instruction list;
4) pseudocode (a design oriented, codelike language limited syntactic and semantic restrictive [23] or cod

The procedure for developing a software design using method is relatively simple.

1) Identify all input data entities. Their format is a seco ary issue to that of the relationship they have to one anoth

2) Organize the input data into a hierarchy.

3) Assign a description of each entity in the input file note the number of times it occurs (e.g., 1 employee fi employees, each employee record contains 5 entries: so security number, hire date, employee number, comp address/location, and home address).

4) Perform the same process as in 1)–3) above but output data.

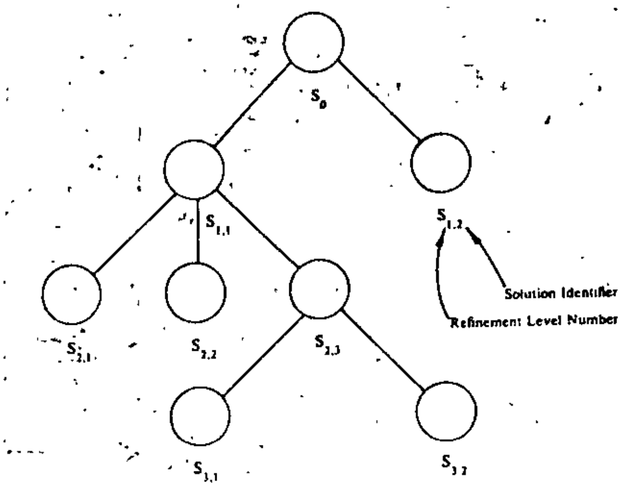


Fig. 12. An overview of iterative stepwise refinement.

5) Specify functional characteristics of the program by defining the instructions (by type) which will occur in the program. These are defined by type of instruction in the following order: read instructions, branch (and related) instructions, computations, outputs, and calls to subroutines.

6) Using a modified flowchart, display the logical sequence of instructions using symbols to represent begin process, end process, branching, and nesting.

7) Number each element of the logical sequence and expand it (where possible) using combinations of the basic set of instructions defined in 6) above.

Warnier provides a number of other more detailed instructions for the composition of programs using this method but these are not germane to our discussion. The development flow using this method is shown in Fig. 11.

C. Prescriptive Methods

This category contains most of the software composition methods currently available [11]. These methods are based on the experience(s) of their authors and lack some unifying conceptual framework. They are the result of valuable experience and a good deal of empiricism. Many of them are a collection of "good things to do" while others put forward an overall approach to solving software design problems. Most have adopted some specific set of design representation schemes. META stepwise refinement [24] does not contain a specific design representation scheme but does provide an overall approach to problem solving. It consists of three concepts: disciplined (or structured) programming, problem solving by iteration, and return on investment/cost effectiveness. The way in which each of these concepts manifests itself in this approach is described below.

1) Disciplined programming–This refers to both the process of decomposing the solution space in order to identify alternative solutions and to the breaking down of a given solution into its constituent attribute. It is not decomposition in the classic sense of the word, as we shall see.

2) Problem solving by iteration–Here, this refers to both the process of repetitively solving a design problem and to the practice of adding detail with each iteration.

3) Return on investment and cost effectiveness–The time and energy necessary to apply the other two concepts are brought into sharp focus via this one. In this case, further refinement is only done on the most promising of the alternative solutions. This makes for a more prudent and cost-effective expenditure of resources.

Finally, the refinement discipline is directed at postponing implementation decisions until the appropriate level of detail has been reached.

This technique begins with the (presumed) existence of an exact stable problem definition. Given that such an expression of the problem exists, the first step is to compose a simple statement of the solution to the problem. Next, several solutions are generated. Each of these is of the same level of detail as the others but each is more detailed than the initial one. Each set of equivalent solutions constitute a level of refinement. The "best" of these solutions is selected. It is now treated in the same manner as the initial solution. That is, several equivalent solutions of more detail are generated, and so forth. The process ends when a level of refinement is reached which can be directly implemented into the intended programming language (Fig. 12).

D. Comments Regarding Software Composition

Each of the methods available today relies heavily on a specific, somewhat inflexible model of software development. Often, the model is localized in character in that it addresses implementation rather than architectural problems. However, the overall effect of the availability of methods in software design has been quite beneficial. Discipline, organization, and an openness in development are a few of the positive trends experienced thus far.

V. Conclusions

Software design is a branch of software engineering which is just beginning to show recognition of its commonality with other design efforts dating back to antiquity. The lack of a standardized approach to depicting software design can be both an asset and a liability. It is an asset in that the means of depicting a design can be tailored to the audience and the type of problems being considered. It is a liability in that much of this potential is lost due to local, factionalized views or requirements on how "best" to portray software–any software. Hopefully, the trend to expand local, program-oriented thinking into a global or systematized approach will accelerate progress toward viewing software design as a *system* of concepts and techniques, and not a secret.

References

[1] G. Mie, "Contributions to the optics of turbid media, especially colloidal metal solutions," *Ann. Phys.*, vol. 25, p. 377, 1908 (in German).

[2] L. W. Carrier, G. A. Cato, and K. J. von Essen, "The backscattering and extinction of visible and infrared radiation by selected major cloud models," *Appl. Opt.*, vol. 6, p. 1029, 1967.

[3] L. J. Peters and L. L. Tripp, "Design representation schemes," presented at the Microware Research Institute Symp. Computer Software Engineering, New York, Apr. 1976.

[4] L. R. Scott, "An engineering methodology for presenting software functional architecture," in *Proc. 3rd Int. Conf. Software Engineering* (Atlanta, GA), May 1978.

[5] L. C. Carpenter, A. S. Kawaguchi, L. J. Peters, and L. L. Tripp, "Systematic development of automated engineering systems," presented at the 2nd Japan-U.S.A. Symposium Automated Engineering Systems, Aug. 1975.

[6] L. L. Constantine, "Structure charts–A guide," course handout from "Structured design course," conducted Nov. 1974.

[7] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 2, pp. 115–139, 1974.

[8] E. N. Yourdon and L. L. Constantine, *Structured Design*. New York: Yourdon Inc., 1975.

[9] SofTech Inc., "How to increase programmer productivity through software engineering using PL/I" (course notes), SofTech, Wal-

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

DIAGRAMAS WARNIER-ORR
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

vi. DIAGRAMAS WARNIER-ORR

(DOCUMENTACION COMPLEMENTARIA)

2

Traditionally, each programmer has applied his own set of rules to the construction of the logic of his program. He starts with this logic structure and, as he encounters additional combinations of conditions to be met, he adds them as afterthoughts rather than revising the logic of the program. The resultant control code might look like that shown in the left (Unstructured) column of Figure 1. This code contains a large number of GO TO statements and labels and its logic is not easy to follow. During subsequent unit and integration testing, disintegration of the programmer's original structure occurs as new constraints and conditions are imposed upon it—leading to more GO TO statements, more labels, and a final program whose original logic may be completely obscured. Reading, understanding, and testing such programs is difficult. The degree of confidence in their quality or correctness tends to be low. In addition, such programs tend to be difficult to maintain and modify.

UNSTRUCTURED

```
          IF p GOTO label q
          IF w GOTO label m
          L function
          GOTO label k
label m   M function
          GOTO label k
label q   IF q GOTO label t
          A function
          B function
          C function
label r   IF NOT r GOTO label s
          D function
          GOTO label r
label s   IF s GOTO label f
          E function
label v   IF NOT v GOTO label k
          J function
label k   K function
          END function
lable f   F function
          GOTO label v
label t   IF t GOTO label a
          A function
          B function
          GOTO label w
label a   A function
          B function
          G function
label u   IF NOT u GOTO label w
          H function
          GOTO label u
label w   IF NOT t GOTO label y
          I function
label y   IF NOT v GOTO label k
          J function
          GOTO label k
```

STRUCTURED

```
①IF p THEN
   A function
   B function
  ②IF q THEN
    ③IF t THEN
       G function
      ④DOWHILE u
         H function
      ④ENDDO
       I function
    ③(ELSE)
    ③ENDIF
  ②ELSE
     C function
    ③DOWHILE r
       D function
    ③ENDDO
    ③IF s THEN
       F function
    ③ELSE
       E function
    ③ENDIF
  ②ENDIF
  ②IF v THEN
     J function
  ②(ELSE)
  ②ENDIF
①ELSE
  ②IF w THEN
     M function
  ②ELSE
     L function
  ②ENDIF
①ENDIF
 K function
 END function
```

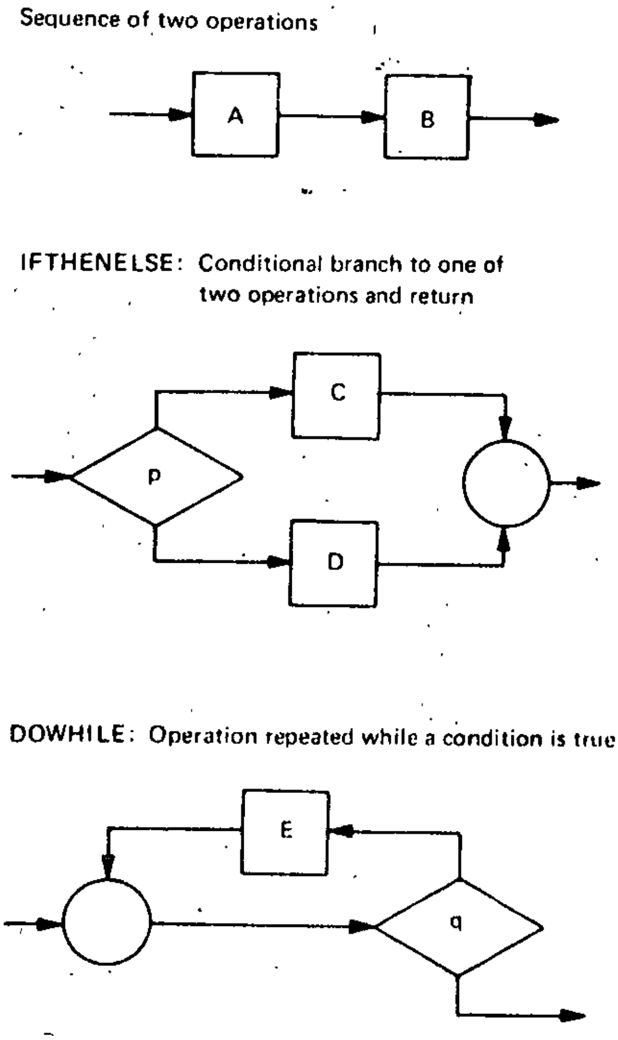
Figure 1. A comparison of structured and unstructured code

Research by computer scientists and mathematicians indicates that an alternative method of programming known as structured programming can help solve these problems. This technique involves coding programs using a limited number of control logic structures to form highly structured units of code that are more readable, and therefore more easily tested, maintained and modified.

Structured Programming Theory

Structured programming is based on a mathematically proven structure theorem[1] which states that any program can be written using only the three control logic structures illustrated in Figure 2:

- Sequence of two or more operations (MOVE,ADD....)
- Conditional branch to one of two operations and return (the IF p THEN C ELSE D of Figure 2)
- Repetition of an operation while a condition is true (the DO E WHILE q of Figure 2)


Sequence of two operations

IFTHENELSE: Conditional branch to one of two operations and return

DOWHILE: Operation repeated while a condition is true

Figure 2. The three elemental logic structures of structured programming

Any program may be developed by the appropriate iteration and nesting of these three basic structures. Each of the three structures has only one entry and one exit. A program consisting solely of these structures is a proper program, a program with one entry and one exit. As illustrated in the structured code (right column) of Figure 1, it always proceeds from the beginning to the end without arbitrary branching. In PL/I, for instance, no GO TO statements are necessary. Proving the logical correctness of structured code is more feasible. The logic is easier to follow, permitting functions to be isolated, understood, and tested.

The use of the three control logic structures in structured programming is analogous to the hardware design practice of forming complex logic circuits from AND, OR, and NOT gates. This practice is based on a theorem in Boolean algebra which states that arbitrarily complex logic functions can be expressed in terms of basic AND, OR, and NOT operations. The use of three control logic structures in structured programming is similarly based on a solid theoretical foundation.

Extensions to the three basic logic structures are permitted as long as they retain the one-entry, one-exit property. An example of such an extension is the DOUNTIL structure (Figure 3), which provides for the execution of the function F until a condition is true.

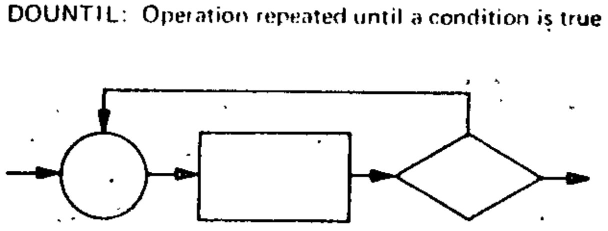

DOUNTIL: Operation repeated until a condition is true

Figure 3. The DOUNTIL structure

Structured Programming Practices

Certain practices are followed to support the objective of producing readable, understandable structured programs—programs in which the writers can have a high degree of confidence.

Indenting within control structure blocks to reflect the logic of the program unit is one of these practices, as shown in the example of structured code in Figure 1. As illustrated by the number to the left of the statements, each logic structure nested within another is indented within it. All parts of a logic structure carry the same indentation level, and functions performed within a logic structure are indented within that structure. This practice highlights the logic of the unit for the writer and the reader and thus contributes to the goal of more readable programs.

- Limiting a unit of source code to a specified size—often one listed page, or fifty lines, permits the programmer to read and understand an entire logical expression or function without referring to multiple pages or relying on his memory. Should the complete logical expression require more than 50 lines of source code, the programmer can segment the code through the use of such statements as %INCLUDE (in PL/I) or COPY (in COBOL) to specify the inclusion of another unit of code (see Figure 4).

```
IF p THEN
   A function
   B function
   IF q THEN
      INCLUDE t-test
   ELSE
      C function
      DOWHILE r
        D function
      ENDDO
      CALL
   ENDIF
   IF v THEN
      J function
   ELSE
   ENDIF
ELSE
   IF w THEN
      M function
   ELSE
      L function
   ENDIF
ENDIF
K function
```

```
t-test

IF t THEN
   G function
   DOWHILE u
     H function
   ENDDO
   I function
ELSE
ENDIF
```

Figure 4. Segmentation

A graphic example of a program constructed of such units is shown in Figure 5. At the top are the job control and linkage editor statements that define the environment and major functions of the program. Subordinate to them is the hierarchy, or calling sequence, of the supporting units of code. Each unit specifies the invocation of the units immediately subordinate to it. Thus unit A would invoke units B and J, using COPY, CALL, PERFORM or %INCLUDE statements; unit B would invoke C and F; unit C would invoke D and E; etc. The next technique to be described, top-down program development, assumes such a hierarchical structure.

[1] Bohm, C., and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM* 9, No. 3 (May 1966), 366-371.

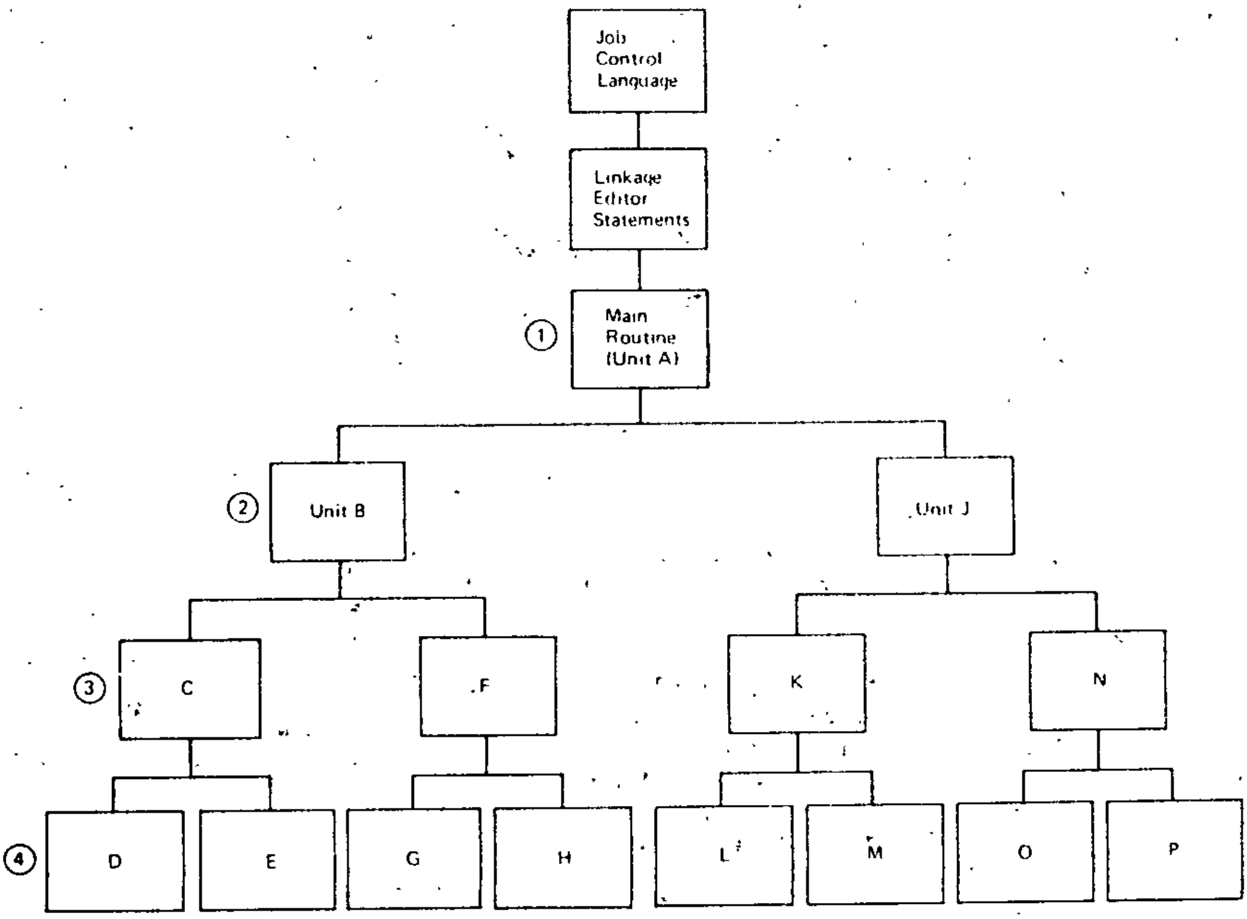


Figure 5. An hierarchical program structure

Chapter 1: An Overview of Structured Programming

Introduction

This chapter contains various items of background information about structured programming that should be useful. The topics include:

Definitions
A summary of the potential advantages of structured programming
The relationship of structured programming to other improved programming technologies
A sketch of the theoretical foundation of structured programming
The basic control logic structures
Additional control logic structures
The GO TO question
Segmentation
Indentation
Documentation considerations
Efficiency considerations
Getting started in structured programming

After you have read the material in this overview, you will be ready to study the reference material in Chapter 2 to see how structured programming can be done in COBOL, and to see some of the ideas illustrated in two sample programs in Chapter 3.

Definitions

Structured programming is a style of programming in which the structure of a program (that is, the interrelationship of its parts) is made as clear as possible by restricting control logic to just three structures:

1. Simple *sequence* of functions
2. *Selection* of functions (IFTHENELSE)
3. Loop control, or *iteration*

These three types of control logic structures may be combined to produce programs to handle any information processing task. Statements controlled by the selection and loop structures are indented to make obvious the scope of influence of the structure.

A structured program is composed of *segments*, which may range from a few statements up to about a page of code. Each segment is allowed to have just one entry and one exit. Such a segment, assuming it has no infinite loops and no unreachable code, is called a *proper program*. When proper programs are combined using the three basic control logic structures (sequence, selection, and iteration), the result is also a proper program.

An important characteristic of a structured program is that it can be read in sequence, from top to bottom, without a great deal of the "skipping around" through the program that is typical of other programming styles. This is important because it is much easier to comprehend fully what a function does if all the statements that influence its action are physically close by. Top-down readability is one consequence of using only three control logic structures, and of avoiding the GO TO statement except in very special circumstances, such as the simulation of a control logic structure in a programming language that lacks it.

A program written according to these principles not only *has* a structure, it clearly *exhibits* it.

Potential Advantages

A program written in this style tends to be much easier to understand than programs written in other styles. Easier understandability facilitates code checking and thus may reduce the program testing and debugging time. This is true partly because structured programming concentrates on one of the most error-prone factors in programming, the logic.

A program that is easy to read and which is composed of well-defined segments tends to be simpler, faster, and less expensive to maintain. These benefits derive in part from the fact that since the program is to a significant extent its own documentation, the documentation is always up to date; this is seldom true with conventional methods.

Structured programming offers these benefits, but it should not be thought of as a panacea. Program development is still a demanding task requiring skill, effort, and creativity.

Relationship of Structured Programming to Other Improved Programming Technologies

Structured programming is compatible with, and supportive of, other improved programming technologies, although distinct from them. Other technologies and the relationship of structured programming to them may be sketched briefly.

Top-down program development involves writing and testing the highest-level segments of a program first, in contrast to the more common method in the past, bottom-up development. This approach has the benefits of giving the critical top segments the most testing, of giving earlier warning of problems with the interfaces between segments, and of spreading the debugging and testing over a greater part of the development cycle.

Structured programming and top-down program development both emphasize the importance of segments that interact in precisely understood ways. Both involve looking at a program as a hierarchy of segments that are related to each other in a tree-like fashion.

Hierarchy plus Input-Process-Output (HIPO) is an approach to functional specification and documentation of programs. Each function is designed using a HIPO diagram, in which inputs and outputs are listed and the processing that is to be carried out is specified. A visual table of contents diagram points to the HIPO diagrams in the package and therefore shows the functions and subfunctions to be carried out by the various parts of a program, and the relationship between them. At the detailed design level, it also shows the hierarchy of segments.

Structured programming, as the term is used in this publication, refers primarily to the coding phase rather than the design phase of the program development cycle. HIPO is one good way to approach the design task, and one that is complementary to structured programming.

A *structured walkthrough* is a review session in which the originator of program design material or code explains it to colleagues. The intent is to detect errors (which are corrected after the walkthrough) as early in the process as possible, when they are least expensive to correct.

Structured programming, with its emphasis on easy readability of programs, increases the effectiveness of structured walkthroughs.

A *development support library* consists of a machine-readable library which contains the current versions of all project programming data. It also consists of external library binders which contain current listings of all library members and archives consisting of recently superseded listings. Besides providing easy accessibility of materials, this helps assure that the latest versions of programs are always used.

Structured programming, with its insistence on segmentation of programs, fits in well with development support libraries, although such libraries are useful with any style of programming.

The *chief programmer team concept* involves programming with teams of at least three members: chief programmer, backup programmer, and program librarian. The team may also include other programmers, nonprogramming analysts, and end users. The chief programmer is responsible for the design and coding of all programs produced by the team, either writing or personally checking every piece of code. The program librarian maintains the development support library.

Structured programming is well-suited to chief programmer team methods, since it facilitates one key element, that of code review by the chief programmer.

Structured Programming Theory

The Structure Theorem

The structure theorem states that any *proper program* can be written using only the control logic structures of *sequence, selection* (IFTHENELSE), and *iteration.*

A *proper program* is defined as one that meets the following two requirements

1. It has exactly one entry point and exactly one exit point for program control.

2. There are paths from the entry to the exit that lead through every part of the program; this means that there are no infinite loops and no unreachable code. This requirement is, of course, no restriction, but simply a statement that the structure theorem applies only to meaningful programs.

The three basic control logic structures are defined as follows:

Sequence is simply a formalization of the idea that unless otherwise stated, program statements are executed in the order in which they appear in the program. This is true of all commonly used programming languages; it is not always realized that sequence is in fact a control logic structure. In flowchart terms, sequence is represented by one function after the other, as shown in Figure 1.

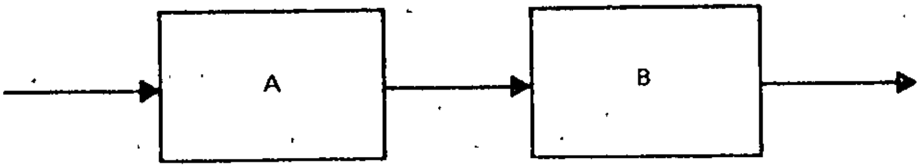


Figure 1. Flowchart for the control logic structure *sequence*

A and *B* are anything from single statements up to complete modules; the concern is only with the abstract idea of a proper program, regardless of its size and internal complexity. *A* and *B* must both be proper programs in the sense just defined (one entry and one exit). The combination of *A* followed by *B* is also a proper program, since it too has one entry and one exit. This can be shown pictorially, as in Figure 2, where the outer box is meant to suggest that the combination of *A* followed by *B* can be treated as a single unit for control purposes.

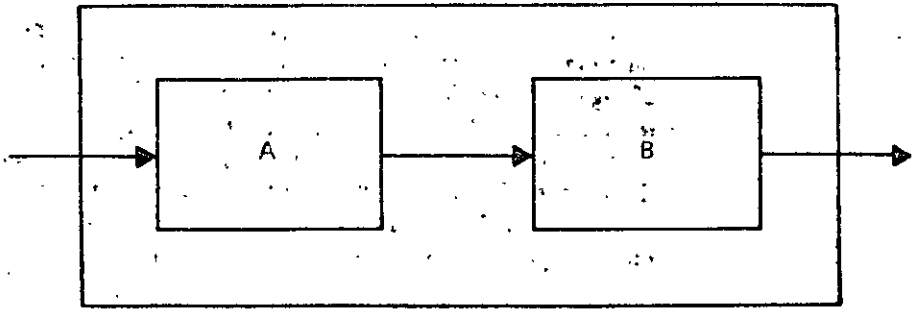


Figure 2. Two proper programs in sequence

Selection is the choice between two actions based on a *predicate:* this is called the IFTHENELSE structure. In COBOL it is implemented with the IF statement, and the predicate is called the *condition*. The usual flowchart notation for selection is shown in Figure 3, where *p* is the predicate and *A* and *B* are the two functions.

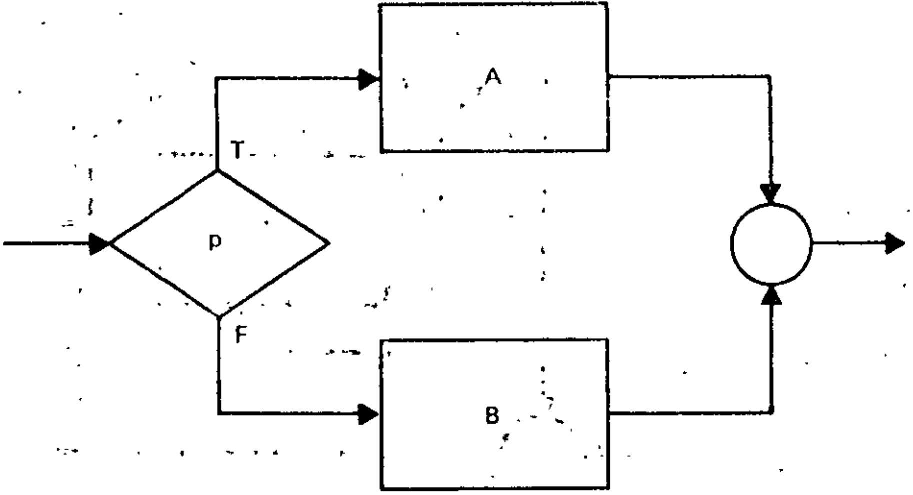


Figure 3. Flowchart for the control logic structure *selection*

The *iteration* structure, used for repeated execution of code while a condition is true (also called loop control), is the DOWHILE. In the flowchart in Figure 4, *p* is the predicate and *A* is the controlled code. In COBOL, the DOWHILE is implemented with the PERFORM . . . UNTIL, as discussed in Chapter 2.

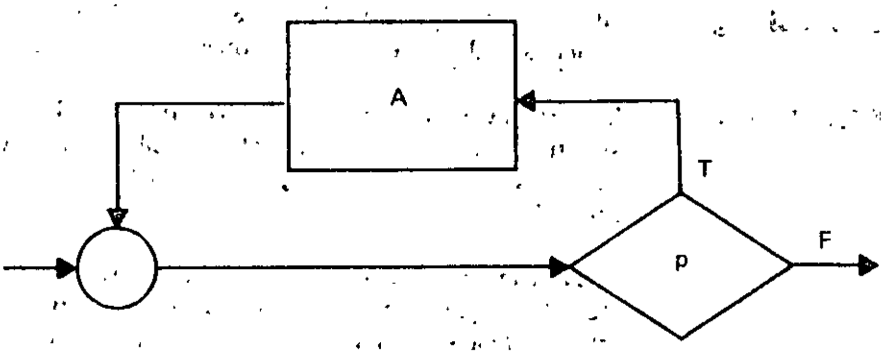


Figure 4. Flowchart for the control logic structure *iteration*, the DOWHILE

A fundamental idea is that anywhere a function box appears, any of the three basic structures may be substituted and still have a proper program. For example, the function box in Figure 4 could be replaced with *selection*, producing the flowchart of Figure 5. The dotted lines show where another structure has been substituted for a function. Or, one function in a *selection* might be replaced with three functions in sequence, and the other replaced with an *iteration*, producing the flowchart of Figure 6. Flowcharts of arbitrary complexity can be built up in this way. Figure 7 shows a flowchart with several control logic structures, drawn this time in top-to-bottom fashion. Two other examples appear in Chapter 3.

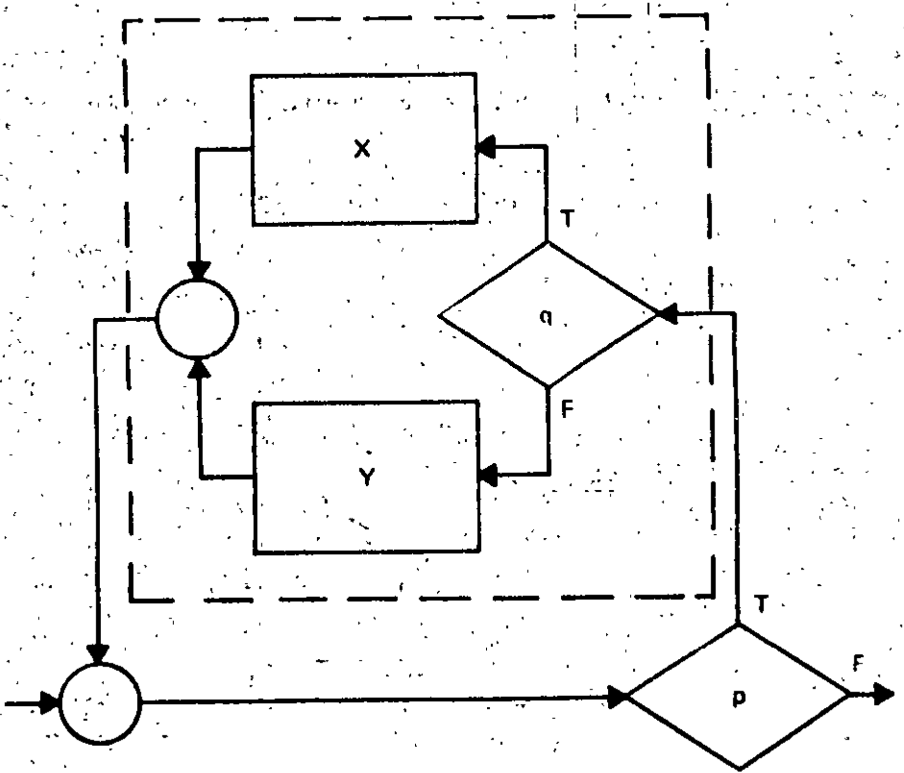


Figure 5. An example of the combination of two control logic structures, in which the function controlled by a DOWHILE is an IFTHENELSE

The ability to substitute control logic structures for functions and still have a proper program is basic to structured programming. This may also be called the *nesting* of structures.

Additional Control Logic Structures

Although all programs can be written using only the three basic structures, it is sometimes helpful to utilize a few others.

The DOUNTIL Structure

The basic iteration structure is the DOWHILE, but there is a closely related structure, DOUNTIL, that is sometimes used, depending on the procedure that is to be expressed and on availability of appropriate language features. The flowchart is shown in Figure 8.

The difference between the DOWHILE and DOUNTIL structures is that with the DOWHILE the predicate is tested *before* executing the function; if the predicate is false, the function is not executed at all. With the DOUNTIL, the predicate is tested *after* executing the function; the function will always be executed at least once, regardless of whether the predicate is true or false.

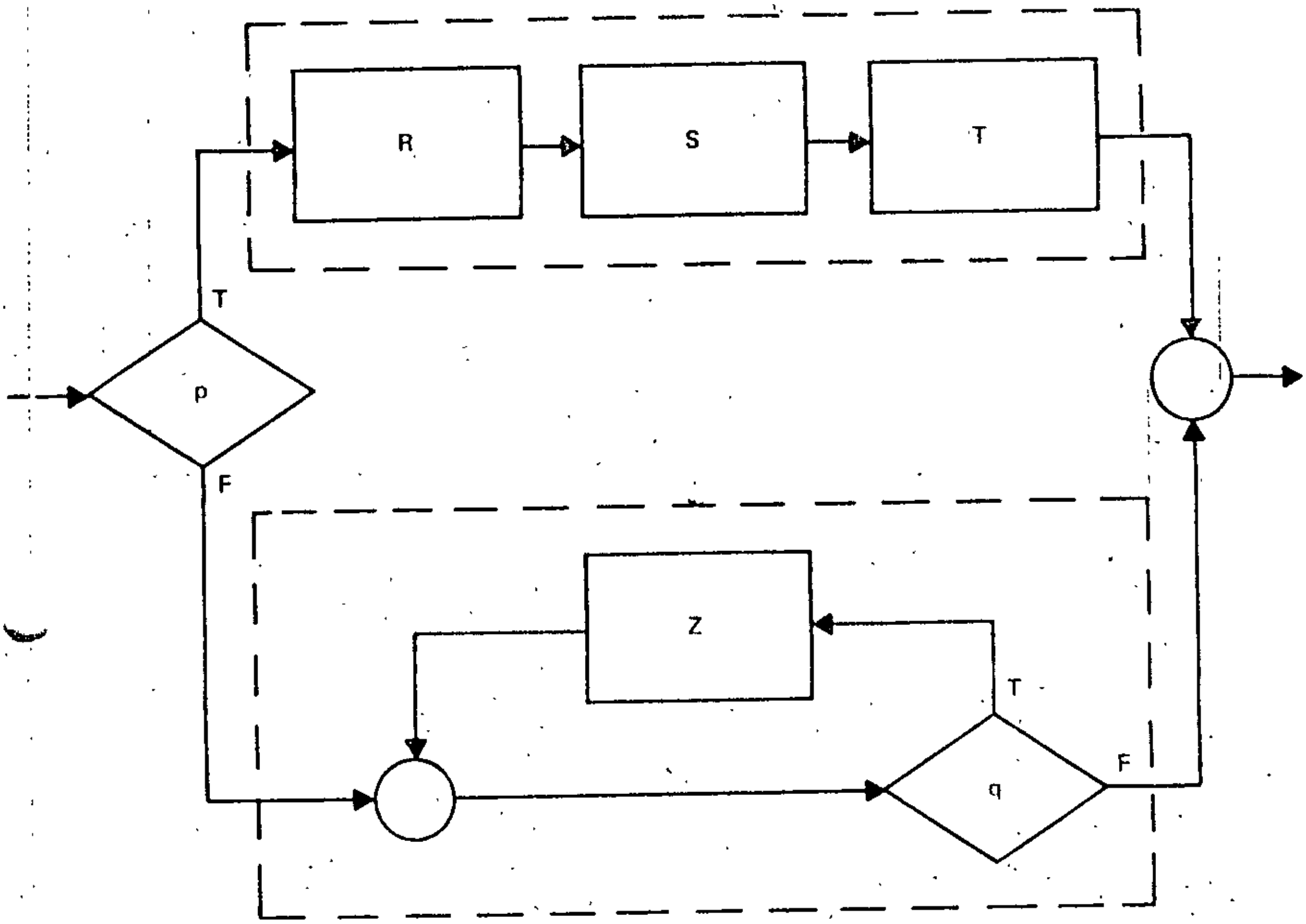


Figure 6. An example of the combination of control logic structures in which a *sequence* and a *iteration* are controlled by a *selection*

The CASE Structure

It is sometimes helpful – from both readability and efficiency standpoints – to have some way to express a multiway branch, commonly referred to as the CASE structure. For example, if it is necessary to execute appropriate routines based on a two-digit decimal code, it certainly is possible to write 100 IF statements, or a compound statement with 99 ELSE IF's, but common sense suggests that there is no reason to adhere so rigidly to the three basic structures.

The CASE structure uses the value of a variable to determine which of several routines is to be executed. The flowchart is shown in Figure 9. Observe that DOUNTIL and CASE are both proper programs.

Efficiency and convenience dictate reasonable use of language elements that may carry out logic functions in ways slightly different from those of the three basic structures. COBOL examples include the PERFORM . . . VARYING verb form. This verb is discussed in Chapter 2 under "DOWHILE".

Labels and GO TO Statements

Structured programming has occasionally been referred to as "GO TO less programming". Although it is true that well-structured programs have few if any GO TO statements, assuming an appropriate programming language, the absence of GO TO's can be misinterpreted. It may be well to pause for a moment to put this issue in context.

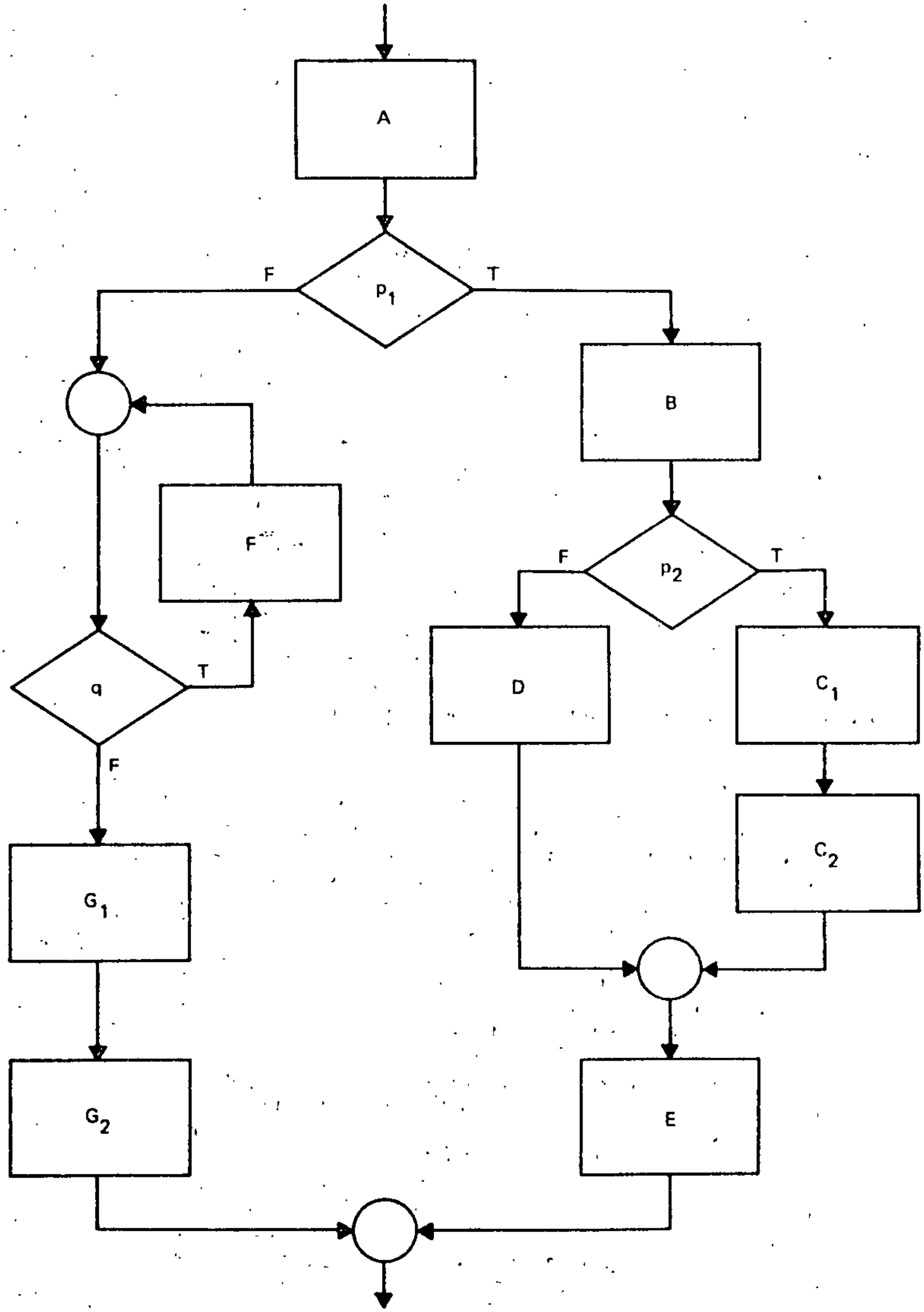


Figure 7. Another example of the combination of control logic structures

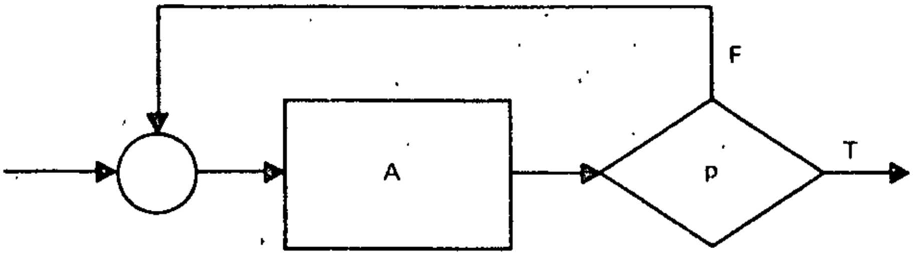


Figure 8. Flowchart for the control logic structure *iteration,* the DOUNTIL

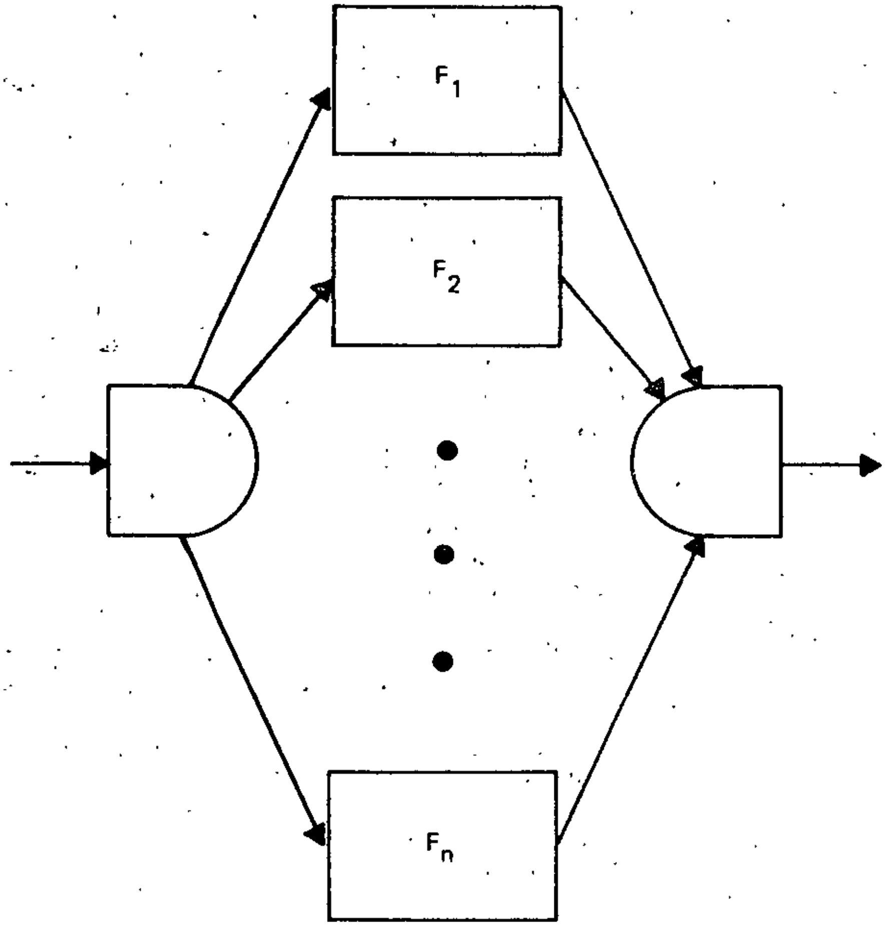


Figure 9. Flowchart for the CASE control logic structure

A well-structured program gains an important part of its easy readability from the fact that it can be read in sequence, without "skipping around" from one part of the program to another. This characteristic is a consequence of the use of only the standard control logic structures (GO TO is not a standard control logic structure). This "sequential readability", or "top-down readability", is beneficial because there is a definite limit to how much detail the human mind can encompass at once. It is far easier to grasp completely what a statement does if its function can be understood in terms of just a few other statements, all of which are physically close by. The trouble with GO TO statements is that they generally defeat this purpose; in extreme cases they can make a program essentially incomprehensible.

No special effort is required to "eliminate GO TO's", which has sometimes been misunderstood as the goal of structured programming. There are indeed good reasons for not wanting to use them, but no extra effort is required to "avoid" them: they just never occur when the standard control logic structures are used. Naturally, if the chosen programming language lacks essential control logic structures, they have to be simulated, and that involves GO TO's. But even this can be done in carefully controlled ways.

There are uncommon situations where the use of GO TO's may improve readability compared with other ways of expressing a procedure. Such examples are exceptional, however, and do not usually occur in everyday programming. The impact of deviations from installation guidelines, such as using GO TO's in other than prescribed ways, should be given careful consideration before such deviations are permitted.

Segmentation

Easy program readability requires that it not be necessary to turn a lot of pages to understand how something works. A practical rule is that a segment should not exceed a page of code, about 50 lines. In COBOL terms a segment can be a paragraph, section, subroutine, or code incorporated with a COPY. (The term segment as used here has nothing to do with the different meanings of the term in connection with the functions of the operating system.)

But segmentation is more than just breaking a program into page-size pieces. What characterizes good program segmentation? Three features can be identified:

1. The segmentation should reflect the division of the program into pieces that relate to each other in a hierarchy, that is, a tree structure. This organization, which may be displayed with a HIPO hierarchy chart, makes it simpler to understand how the segments relate to each other. Further, the segments at the top of the hierarchy should contain high-level control functions, whereas the segments at the bottom should contain detailed functions.

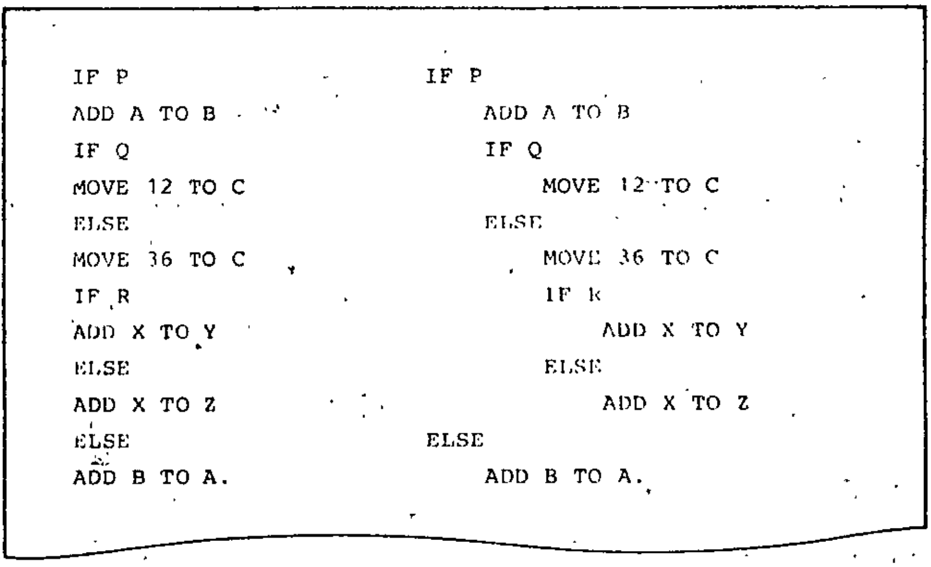
2. A well-designed segment carries out functions that are closely related to each other. This makes it easier to understand and therefore easier to assure that it does what it is meant to do. It also means that when changes have to be made, either during original programmiig or in maintenance, there is less chance of disturbing portions of the program that do not change.

3. A well-designed segment communicates with other segments only in carefully controlled ways. Some proponents of structured programming urge that segments always consist of subroutines, so that the only possible communication between segments is through the subroutine parameter list (the USING clause, in COBOL); this reduces the chance that segments will interact in unintended and undesirable ways. This is too restrictive for general use in COBOL programming, where segments will also consist of paragraphs, sections, or code incorporated through a COPY. Such segments communicate through the mechanism of using the same data items in the Data Division.

Indentation

The use of indentation is important because consistent indentation enhances readability so that the finished program exhibits in a pictorial way the relationships among statements. The basic idea is that all the statements controlled by a control logic structure should be indented by a consistent amount, to show the scope of control of the structure. In COBOL this means that the statements between the IF and the ELSE should be indented a consistent amount, and similarly for the statements between the ELSE and the next sentence. In COBOL there is no language element for iteration that permits the controlled code to be inline. The paragraph controlled by a PERFORM must be placed out of line from the paragraph in which the PERFORM appears.

Indentation can be a major benefit, as the skeleton programs in Figure 10 show. Both do the same processing, but the second is far easier to understand and, therefore, to verify for correctness.



```
IF P                    IF P
ADD A TO B                  ADD A TO B
IF Q                        IF Q
MOVE 12 TO C                    MOVE 12 TO C
ELSE                        ELSE
MOVE 36 TO C                    MOVE 36 TO C
IF R                            IF R
ADD X TO Y                          ADD X TO Y
ELSE                            ELSE
ADD X TO Z                          ADD X TO Z
ELSE                    ELSE
ADD B TO A.                 ADD B TO A.
```

Figure 10. A nested IF statement, with and without indentation

Establishing Indentation Guidelines

Guidelines for indentation in COBOL programs are suggested in Chapter 2. It should be understood, however, that these are *only* guidelines. Each installation will need to establish local conventions; variation from the suggested guidelines is not important, so long as the installation conventions are followed consistently. For example, it is not of fundamental importance whether the statements controlled by an IF are indented four spaces, or three, or two. Arguments can be made for each, but there is no one way that is absolutely right. Within any one installation, however, *some* set of rules should be followed, or the value of indentation will be lost.

Creating a Structured Program

Structured programming, as the term is used in this publication, refers to the coding portion of the total program development cycle. It may help to sketch the cycle, indicating how structured programming relates to each phase.

The program development process can begin when, in response to a statement of requirements, a *specification* is developed that states the objectives of the application. Then, initial design takes place, during which each major function is identified and then subdivided into lower level functions. HIPO diagrams are a design aid and documentation tool in this stage. It is important in initial design not to become enmeshed in low level details; the strategy is to manage complexity by attacking the problem one level of detail at a time.

It is not to be expected that program design will proceed in a straight-line fashion. The HIPO hierarchy chart may have to be drawn several times, as the expected segment size or the implications of logic flow become clearer. The basic idea is to begin with a top-level attack, with little detail, then fill in the successive levels, refining original plans as necessary until the design is complete.

Once the initial design is complete, programmers refine the design to add the details necessary for the coding process. In *detailed program design,* additional HIPO diagrams are created to specify further detail about each process. If flowcharts are used to express logic flow, they should include only the basic structures. Another technique used in the detailed design phase is pseudocode, an informal means of expressing logic. Although HIPO diagrams can reduce the need for other documentation of logic flow, flowcharts and pseudocode can be used with HIPO diagrams.

In pseudocode, the basic control logic structures and indentation are used in a carefully controlled way, but everything else is at the discretion of the programmer: elements of programming languages may be utilized, or mathematical notation if it is appropriate to the application, etc. Pseudocode is similar to a programming language, but it is not compilable, and it is not bound by formal syntactical rules. Pseudocode is used to depict detailed logic while avoiding the distractions of the details of programming language requirements; it is easier to modify than programming language statements. When detailed program design is finished, the translation from pseudocode to the chosen programming language is straightforward, since the most difficult part (the logic) is finished. An example of pseudocode appears in an illustrative program in Chapter 3.

In the *coding* stage of program development, the techniques that have become identified with structured programming, as the term is used here, come into greatest prominence. Program statements implementing control logic structures are used, and they are indented to show the scope of influence of the structures; thus, the details of code are clearly related to the structure of the design. For ease of understanding, no structure is allowed to extend over a page boundary. Meaningful data and paragraph names are used, perhaps following conventions that suggest the functions of the data and paragraphs. Program segments are proper programs (one entry, one exit), and can be read in sequence from top to bottom.

It is becoming increasingly common for completed code to be checked by another programmer, either in a structured walkthrough or in some other kind of code reading process. During *test* program errors are located and it is verified that the program performs according to specifications. With structured programs this stage may tend to take less time than before because errors can be located and corrected more rapidly in the more readable structured code.

Finally, the program has to be *maintained* over the period of its use. Specifications change, equipment configurations are modified, and coding errors are discovered; these may require program modifications. Over the life of a major program, maintenance often requires more effort than the original program development.

Structured programming facilitates program maintenance for much the same reason that it facilitates program testing: the program is easier to understand. Whether the original programmer or a different maintenance programmer is involved, changes are easier to make and are less likely to cause undesired effects elsewhere in the program.

In summary, program development consists of requirements, specification, initial design, detailed design, coding, test, and maintenance. The most difficult task is design, which properly should receive the most attention and effort, since errors here are least costly to correct.

Documentation

How much documentation of a program's logic is needed in addition to the program itself? In the past it has sometimes been argued that the logic of a program should be documented with a complete set of flowcharts. This contention may need to be reevaluated for structured programs, which display their own logic better than conventional programs.

To reduce the need for documentation of logic, the code should follow certain guidelines of good programming practice that for many years have been characteristic of the best programmers. Data names and labels should be as indicative as possible of the functions of the data items and program elements, even if this tends to lengthen the names. "Tricky" coding should always be avoided.

When these and other common-sense principles have been followed, and the program has been written according to the principles of structured programming (only a few control logic structures, indentation, top-down readability), there will be little need for documentation of the logic flowchart type. (The need for documentation of function provided by HIPO hierarchy charts and HIPO diagrams, such traditional documentation as data layout charts, as well as data preparation instructions, etc., is not affected by structured programming.)

Efficiency Considerations

Programmers are sometimes concerned that structured programming techniques may lead to object programs which run slowly or which create problems in a virtual storage system. There is nothing inherent in the structured programming approach that leads to inefficiency; the use of a restricted set of control logic structures and of segmentation does not automatically carry any time or space penalty.

Although no systematic study of many users has been made, some users have reported that structured programming techniques usually lead to no performance penalties. In a particular language there may be some overhead, however, as in COBOL with the heavy use of the PERFORM verb that is normal with structured programming. Problems, when and if they occur, should be seen in context of the full range of considerations that determine the effectiveness of a data processing operation. For instance, the ability to create programs on time may be much more important than a small object program speed penalty. Or, it may be noted that a "highly efficient" program that is very difficult to maintain is not really efficient in the context of total cost. Finally, efficiency always relates to a specific environment of compilers, hardware, and user code.

If object program speed *does* become a problem, however, the following approaches may be considered.

Identify those portions of the program that are most heavily used; various analysis programs may be helpful in doing so, such as by providing counts of verb executions. It will usually be found that a rather small part of the program has a large influence on speed. Concentrate on those few segments. It may be necessary to recode PERFORM statements to inline code, or to "unwind" short, heavily used loops. Attention should be given to the possibility of avoiding certain data conversions or language features that may adversely affect performance. Since usually only a small part of the program needs to be modified, this modification will ordinarily not take a great deal of effort.

If excessive paging in a virtual storage system is a problem, the basic solution is to place in the same virtual storage page code segments that are used together. This means rearranging paragraphs in the source program. Again, analysis programs can be a help. Structured programming can actually be a benefit in this kind of tuning, since segments are never entered except by a reference to their names (this makes it easy to move segments in the source program). Further, performance problems, whatever coding techniques are used, can seldom be predicted in advance. Because of the ease of maintaining (changing) structured programs, the likelihood is that performance problems can be more easily corrected.

Getting Started in Structured Programming

One way to evaluate structured programming in an installation is the following:

- Management authorizes the use of structured programming in a project. The first structured programming project should be neither trivial nor extremely difficult, but rather one that would be considered of normal size and level of difficulty. At least two programmers should be assigned to the project so that they can check each other's code.

- Programmers assigned to the project familiarize themselves with the subject. Some installations have implemented structured programming on their own; others have found that attending a class was necessary. Experienced COBOL programmers may be unfamiliar with or reluctant to use the following COBOL language facilities required or permitted in structured COBOL programs:
  - Nested IF's
  - PERFORM UNTIL
  - Compound conditions for nested IF's and PERFORM UNTIL

  Therefore, it may be advisable for programmers implementing structured programming in COBOL without attending a class to review these COBOL statements in the appropriate reference manual.

- A set of guidelines for the initial effort should be established. Those in Chapter 2 on COBOL implementation could be used, many installations will prefer to establish their own. The guidelines for the first project should avoid extending the permissible control logic structures; uncontrolled extensions can easily destroy the value of structured programming. Some programmers find it helpful to summarize the guidelines in the form of a checklist or a simple illustrative program.

- After creating the HIPO diagrams and visual table of contents, pseudocode or flowcharts can be used for detailed logic, if appropriate. The code is then written and the program tested.

The evaluation process can be repeated and the guidelines modified until the programmers have sufficient experience with structured programming. At this time structured programming guidelines can be incorporated into the installation's standards.

Chapter 2: Implementing Structured Programming in COBOL

Introduction

Once the principles of structured programming are understood, writing structured programs in COBOL is a matter of habitually following a few simple rules. All control logic structures can be expressed in COBOL, although in some cases not directly.

The subject can be approached in terms of three major considerations:
1. COBOL implementation of the control logic structures, both basic and extended
2. Organization of a structured COBOL program
3. Indentation conventions

Also of interest are some pointers on naming conventions, use of comments, the ALTER statement, and special conditions.

Basic Control Logic Structures

SEQUENCE

Sequencing is implemented in the COBOL language simply by writing statements in succession.

IFTHENELSE

The IFTHENELSE structure tests a predicate to determine which of two function blocks will be executed.

The *flowchart* of the IFTHENELSE structure is shown in Figure 11.

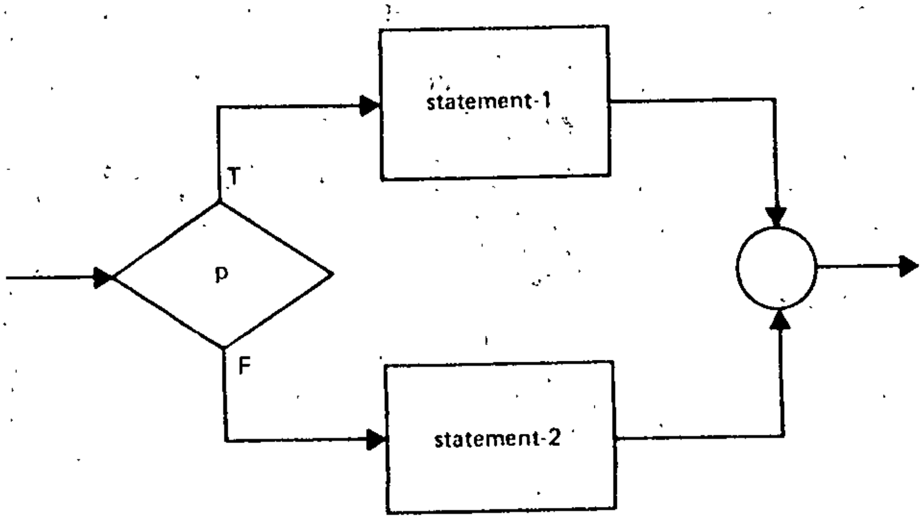


Figure 11. Flowchart for the IFTHENELSE

The *pseudocode* of the IFTHENELSE is:

```
IF condition-p
THEN
    statement-1
ELSE
    statement-2
ENDIF
```

The *COBOL IF statement format* for the IFTHENELSE is:

```
IF condition-p
THEN
    statement-1
ELSE
    statement-2.
```

Statements controlled by the THEN and ELSE parts of the IF are indented to exhibit the span of control of the structure. The THEN and ELSE are started in the same column as the IF. The word THEN is an optional IBM extension of the American National Standard COBOL, X3.23-1968 (formerly known as USA Standard COBOL).

The only COBOL equivalent of the ENDIF operator is the period at the end of the sentence. This means that in a nested IF there is no ENDIF operator for the inner IF statement. Two common control logic structures accordingly require special handling.

A null ELSE within a nested IF, depicted in Figure 12, has the following pseudocode:

```
IF p
THEN
    IF q
    THEN
        F1
    ENDIF
ELSE
    F2
ENDIF
```

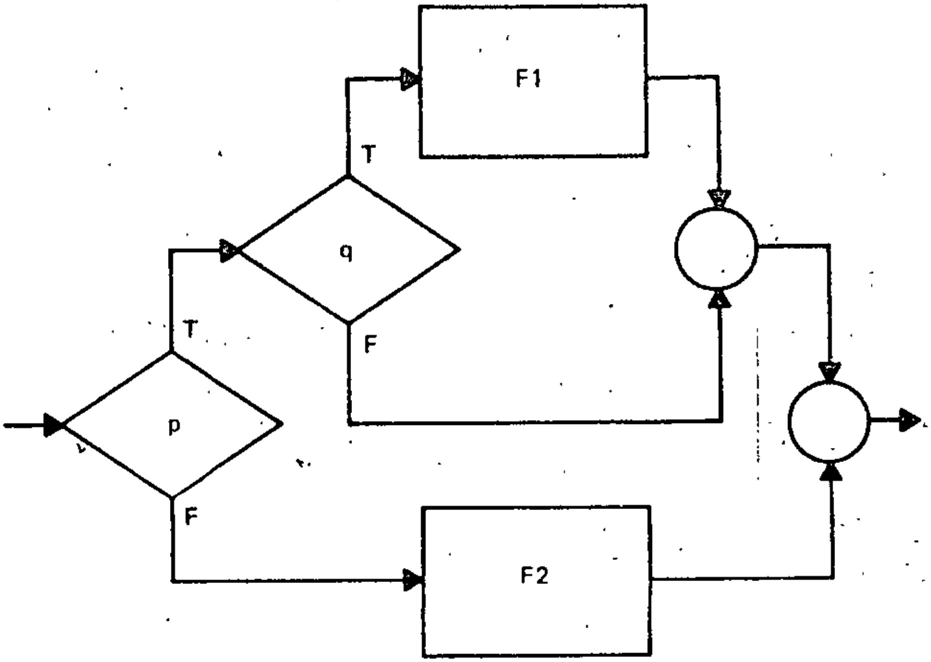


Figure 12. A control logic structure with a null ELSE path within a nested IF

Since an ELSE in COBOL is paired with the most recent IF without a corresponding ELSE, a direct COBOL implementation of the pseudocode would place F2 on the ELSE branch of q, and give p a null ELSE. In this and subsequent examples, "p" and "q" will be used as names of conditions. The problem is readily handled with the NEXT SENTENCE clause:

```
IF p
THEN
    IF q
    THEN
        F1
    ELSE
        NEXT SENTENCE
ELSE
    F2.
```

A function following a nested IF, statement-3 in Figure 13, presents a special problem.

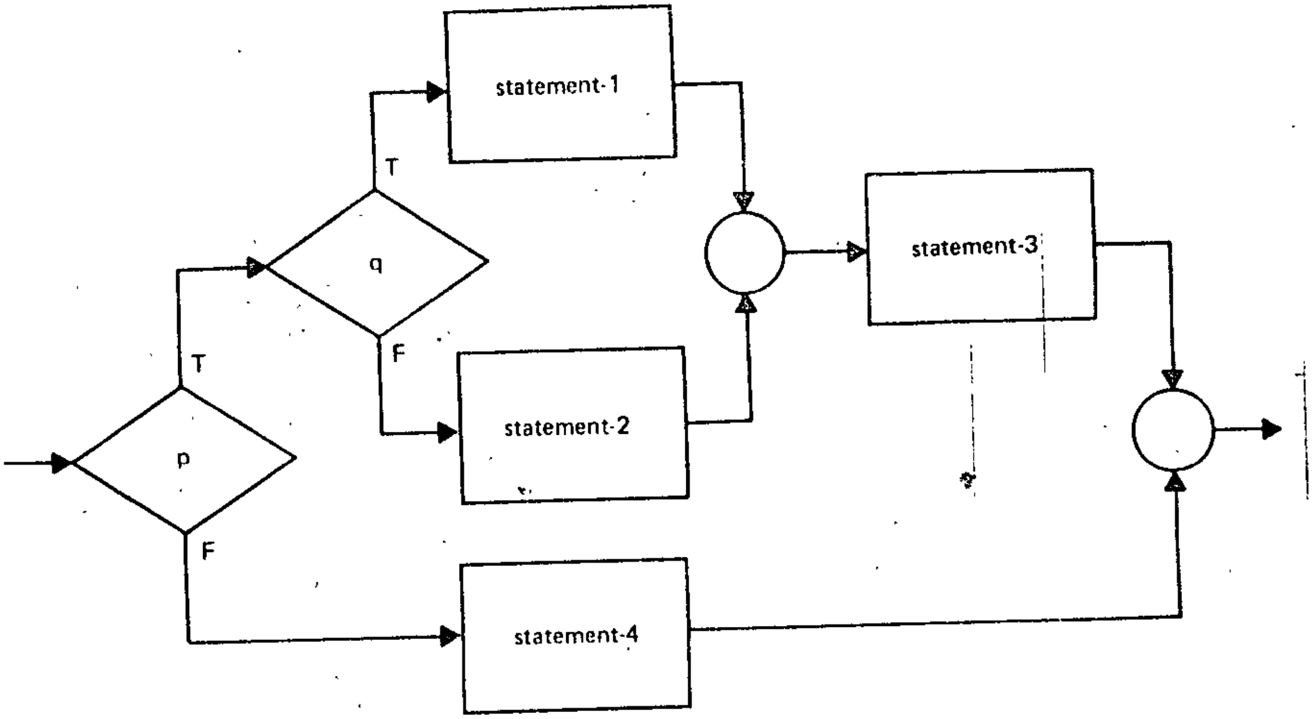


Figure 13. A control logic structure that cannot be directly implemented with a COBOL nested IF

The pseudo code for a nested IF is as follows:

```
IF p
THEN
    IF q
    THEN
        statement-1
    ELSE
        statement-2
    ENDIF
    statement-3
ELSE
    statement-4
ENDIF
```

In COBOL, lacking an ENDIF to terminate the inner IF, a direct implementation would place statement-3 on the ELSE path of the inner IF, rather than making it independent of the inner IF as desired. The simplest solution is to perform the inner IF:

```
IF p
THEN
    PERFORM inner-if
    statement-3
ELSE
    statement-4.
inner-if
    IF q
    THEN
        statement-1
    ELSE
        statement-2.
```

DOWHILE

The DOWHILE structure tests a predicate and executes a function so long as the predicate is true. The flowchart is shown in Figure 14.

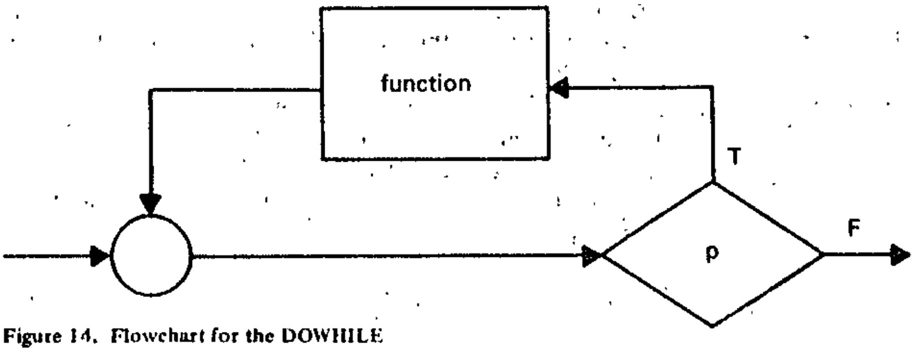


Figure 14. Flowchart for the DOWHILE

The pseudocode for the DOWHILE is:

```
DOWHILE p
    function
ENDDO
```

The COBOL implementation of the DOWHILE involves the PERFORM verb with an UNTIL phrase.

```
PERFORM paragraph-name
    UNTIL (NOT p)
```

The flowchart for the COBOL implementation of the DOWHILE is shown in Figure 15. Since the PERFORM . . . UNTIL repeats execution of the named procedure until the condition is true, the opposite condition is tested, and the true and false labels on the predicate are reversed from the flowchart of the DOWHILE shown in Figure 15. In actual practice, it may be preferable to code the inverse operators. For example, if p is (A IS EQUAL TO B), the inverse (A IS NOT EQUAL TO B) is used rather than (NOT (A IS EQUAL TO B)).

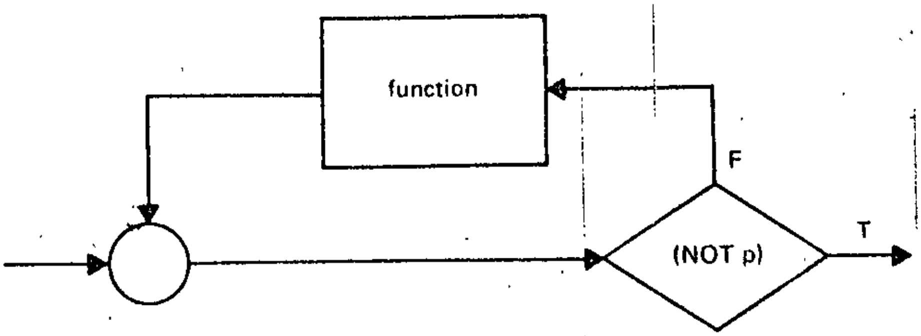


Figure 15. Flowchart representation of the DOWHILE control logic structure as implemented by the PERFORM . . . UNTIL in COBOL

The following form may be used for pseudocode:

```
PERFORM UNTIL (NOT p)

    function

ENDPERFORM
```

COBOL, of course, does not permit the function controlled by the PERFORM to be written inline in this way, but allowing it in pseudocode can help to clarify relationships.

It should be emphasized that the PERFORM . . . UNTIL carries out the test of the predicate *before* executing the controlled function. If the condition in the UNTIL phrase is true when first encountered, the paragraph or section named in the PERFORM is not executed at all.

If indexing is used, the DOWHILE structure may be coded as follows:

```
PERFORM procedure-name

    VARYING identifier-1 FROM identifier-2 BY

    identifier-3

    UNTIL condition-1.
```

DOUNTIL

The DOUNTIL structure executes a function and then tests a predicate to determine whether to repeat it again. The flowchart is shown in Figure 16.

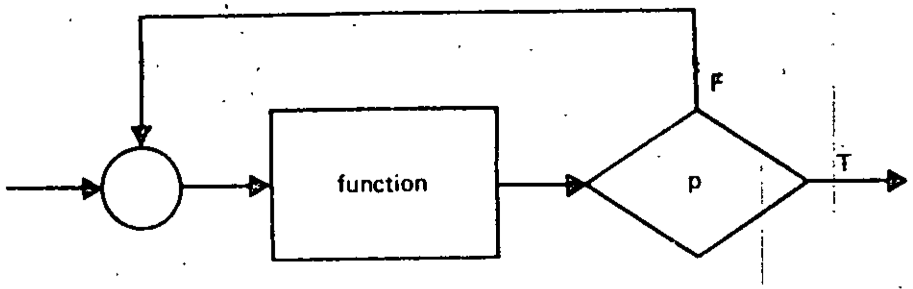


Figure 16. Flowchart for the DOUNTIL

The pseudo code for the DOUNTIL is:

```
DOUNTIL p

     function

ENDDO
```

COBOL has no direct equivalent of the DOUNTIL, but it is rather easily implemented with the PERFORM, in either of two ways. The first way simply starts with an unconditional PERFORM to guarantee that the function is carried out at least once:

```
PERFORM paragraph-name.

PERFORM paragraph-name

     UNTIL p.
```

The second, more cumbersome way is to set a flag to a value before executing the PERFORM, and then conditionally change it within the named paragraph:

```
MOVE 'NO' TO FLAG.

PERFORM paragraph-name

     UNTIL FLAG = 'YES'.

.

.

.

paragraph-name.
```

```
processing statements.
IF condition
THEN
    MOVE 'YES' TO FLAG.
```

CASE

The CASE structure selects one of a set of functions for execution, based on the value of a parameter. The flowchart notation is shown in the example of Figure 17. COBOL offers no direct implementation of CASE, but it may be simulated using a combination of the PERFORM and the GO TO . . . DEPENDING ON statements. Figure 18 shows the COBOL coding for Figure 17. Notice that the PERFORM must be written with the THRU option because the paragraphs that carry out the individual functions all end with a GO TO paragraph-EXIT; without the THRU, this EXIT paragraph would not be correctly interpreted. Observe that the statements following the GO TO . . . DEPENDING ON are to handle the case where ACCOUNT-CODE has a value less than one or greater than the number of paragraphs named. Finally, where "processing statements" are shown, it would also be quite possible to have PERFORM statements invoking out-of-line segments to do the processing.

The CASE statement cannot be implemented in COBOL without using the GO TO statement, but the problems associated with the GO TO are minimized by using it only in a highly controlled way. Someone reading this implementation of the CASE structure can be assured that transfer will be limited to paragraphs within this structure and the processing paragraphs all transfer to the CASE EXIT. The problems associated with "skipping around" while reading the program are thereby avoided.

Program Organization

A structured COBOL program is organized into segments. (This usage of the word "segments" has no relation to the meaning of the COBOL reserved word SEGMENT, or to the uses of the word in connection with the operating system.)

The first segment of the program consists of the Identification and Environment Divisions. The second consists of the Data Division; this is the only segment in the program that may exceed one page in the source program listing, about 50 lines, including blank lines. The Procedure Division is made up of segments consisting of sections and paragraphs.

The first segment of the Procedure Division is the mainline routine. It is the only section or paragraph in the program that is entered other than through the CALL or PERFORM mechanisms (with the exception of the simulation of the CASE structure).

A Procedure Division segment, as the term is used here, consists of one or more paragraphs or sections, or of an entire subroutine. A Procedure Division segment may be invoked with a PERFORM; a segment may be part of a subroutine library and invoked with a CALL; a segment may be included in the program using a COPY. The decision of which to use depends on such factors as the availability of precoded routines, performance considerations, and the system being used.

A segment may consist of more than one paragraph when a subordinate paragraph exists only to implement a control logic structure in COBOL. For example, if the only reason for putting a READ into a separate paragraph is to place it under the control of a PERFORM . . . UNTIL, the separate paragraph may be considered part of the segment that invokes it.

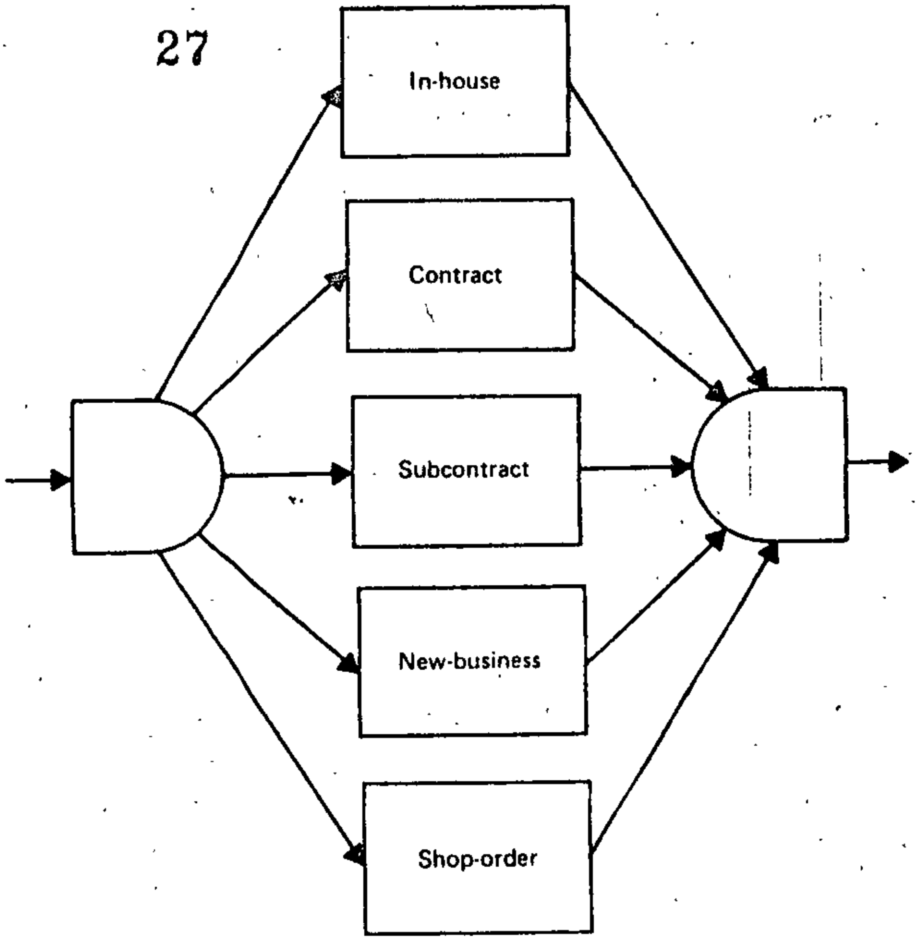
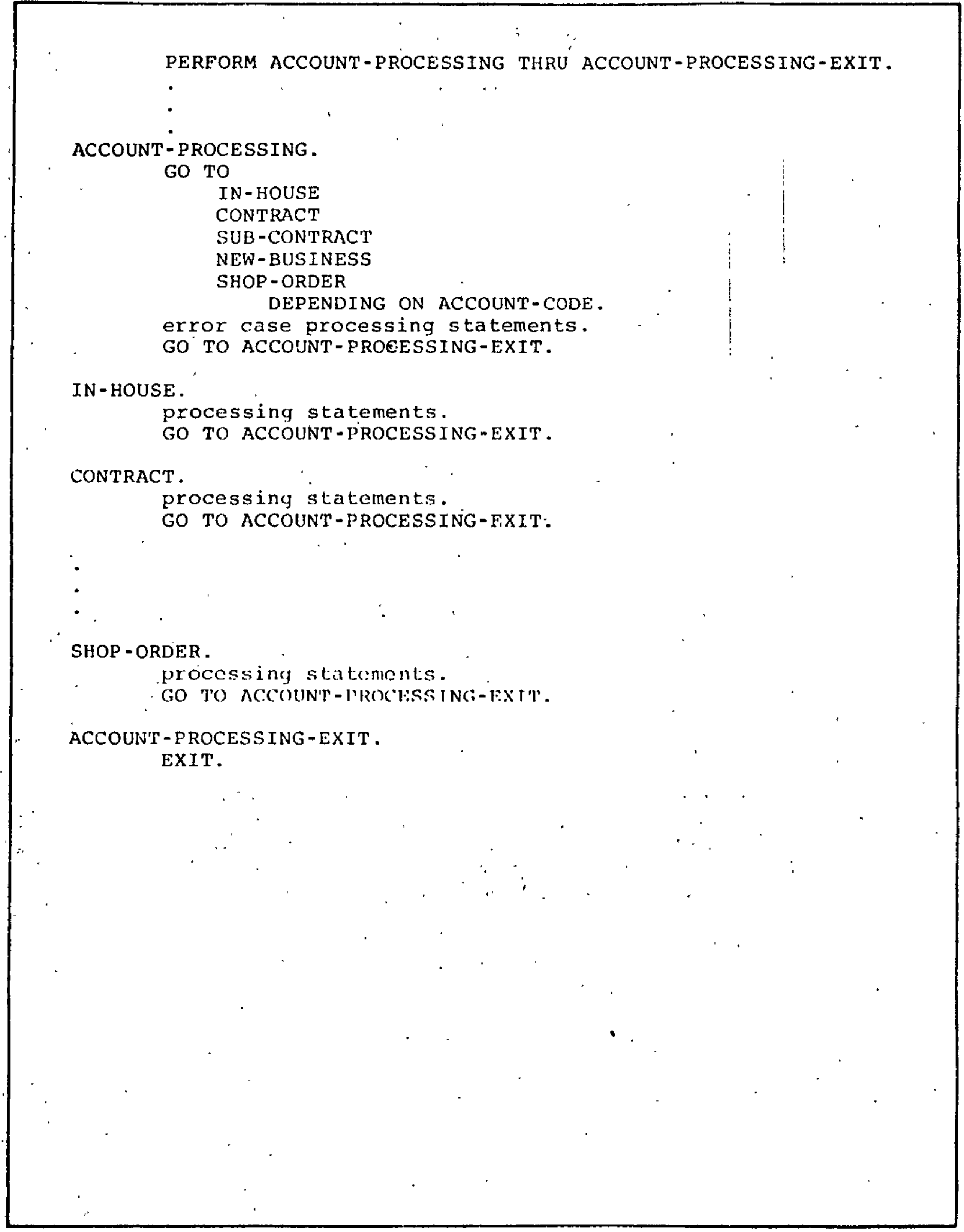


Figure 17. Flowchart for the CASE control logic structure

The EJECT statement can be used to place each segment on a separate page of the program listing. If a program contains many short segments, paper conservation becomes a consideration, and it may be preferable to use the SKIP statement to place a few blank lines between the segments rather than putting each one on a separate page. A segment should still not extend over a page boundary.

The segments of the Procedure Division should appear in some logical sequence, such as the order in which they are invoked, or a sequence that reflects their position in the logical hierarchy of the program. It may be desirable in some cases to affix prefixes to paragraph names so that the names are in sequence by prefix; this makes locating a paragraph in a long program much easier.

When segments are included with a COPY, the source listing does not show the code, which is obtained from a library during compilation. The program is compiled as though the included code were inline at the point of the COPY. A COBOL restriction is that the code obtained with a COPY may not itself contain a COPY; that is, COPY statements may not be nested.



```
       PERFORM ACCOUNT-PROCESSING THRU ACCOUNT-PROCESSING-EXIT.
       .
       .
       .
ACCOUNT-PROCESSING.
       GO TO
           IN-HOUSE
           CONTRACT
           SUB-CONTRACT
           NEW-BUSINESS
           SHOP-ORDER
               DEPENDING ON ACCOUNT-CODE.
       error case processing statements.
       GO TO ACCOUNT-PROCESSING-EXIT.

IN-HOUSE.
       processing statements.
       GO TO ACCOUNT-PROCESSING-EXIT.

CONTRACT.
       processing statements.
       GO TO ACCOUNT-PROCESSING-EXIT.

.
.
.

SHOP-ORDER.
       processing statements.
       GO TO ACCOUNT-PROCESSING-EXIT.

ACCOUNT-PROCESSING-EXIT.
       EXIT.
```

Figure 18. A skeleton program illustrating the implementation of CASE in COBOL

Indentation Guidelines

The following are *only* suggestions. No standardization of indentation conventions has developed so far, and there seems to be little pressure for it so long as consistent standards are followed within any one organization. The reasons given for the suggestions that follow will be a guide in developing installation standards. The sample programs in Chapter 3 illustrate many of the guidelines.

The key idea in devising helpful indentation guidelines is the production of programs in which the visual layout of the program elements aids the reader in understanding program relationships and functioning.

Following are some ways this may be done in the Data Division:

- If all level 01 and level 77 (if any) entries have their level numbers in columns 8-9 and their names starting in column 12, it is easier to see where groupings begin and end.
- Entries are much easier to compare, and record layouts are more readily comprehended, if all PICTURE clauses begin in the same column and are thus vertically aligned. The choice of column depends on the length of the longest data name and on the depth of the level structure; starting all PICTURE clauses somewhere in the area of columns 32-45 would be reasonable.
- When more than one line is needed for a Data Division entry, the fact that the lines after the first are part of the same entry is made clearer if the continuation lines are indented. The number of spaces of indentation is not crucial, so long as it is consistent; four spaces might be suitable.
- Record structure is highlighted by indenting each successive level by some standard amount, such as two or four spaces.
- The purpose of individual data items (that is, those not part of records) is made clearer by grouping them under level 01 entries with descriptive names, such as SUBSCRIPT-ITEMS or ERROR-MESSAGES, rather than making level 77 entries of them.
- The free use of blank lines can exhibit more clearly that relationships exist among items so grouped.

Here are some possibilities for the Procedure Division:

- Paragraph and section names are easier to locate if they are always written on a separate line, that is, if the first statement following a name begins on the next line.
- The statements controlled by the THEN and ELSE parts of an IF statement are shown to be subordinate to the logic by indenting these statements by a consistent amount. If one tends to use nested IF statements sparingly, an indentation unit of four spaces would be reasonable. If one writes deeply nested IF statements, however, the indentation would need to be three or two positions, to avoid running out of space in the line.
- Verbs are easier to locate, and programs are easier to correct and modify, if two statements are never written on the same line.
- Statements are easier to pick out if second and following lines of a continued statement are indented by some consistent amount, such as four spaces.

- Important clauses of certain COBOL verbs are better set off if they are written on separate lines, indented by whatever amount is chosen as the standard for continuation lines. For example, when the UNTIL and/or VARYING options are used with the PERFORM's, they can be written on separate lines and indented:

```
PERFORM LINE-OUT

    VARYING LINE-NUMBER FROM 1 BY 1

    UNTIL LINE-NUMBER IS GREATER THAN 50.
```

The VARYING, AT END, and WHEN clauses of a SEARCH can be written on separate lines and indented:

```
SEARCH RATE-TABLE

    VARYING LINE-IDENTIFICATION

    AT END MOVE S TO NO-FIND-SW

    WHEN MF-RATE = RATE-TABLE (RATE-INDEX)

    NEXT SENTENCE.
```

- Various statements are easier to read if appropriate portions of the statements are aligned vertically:

```
OPEN INPUT  TRANSACTION-FILE

            OLD-MASTER-FILE

     OUTPUT EXCEPTION-FILE

            NEW-MASTER-FILE

            REPORT-FILE.

MOVE T-ACCOUNT-NUMBER TO R-ACCOUNT-NUMBER.

MOVE T-HOURS-WORKED   TO R-HOURS-WORKED.

MOVE T-PAYRATE        TO R-PAYRATE.

MOVE W-GROSSPAY       TO R-GROSSPAY.

MOVE W-NETPAY         TO R-NETPAY.
```

- It is necessary to be careful with indentation, since it is possible to mask mistakes. For example, suppose one wrote this:

```
IF SIZE-CODE = 'K'
THEN
    IF TOTAL-PRICE IS GREATER THAN 500
    THEN
        MOVE CHECK-PRICE-MESSAGE TO OUTPUT-LINE
        PERFORM LINE-OUT
ELSE
    MOVE BAD-SIZE-CODE MESSAGE TO OUTPUT-LINE
    PERFORM LINE-OUT.
```

The indentation conveys a presumed intention to make the ELSE match the first IF – but the COBOL compiler does not recognize indentation conventions, and will match the ELSE with the *second* IF. (Inserting an ELSE NEXT SENTENCE to complete the inner IF will give the action represented by the indentation.)

Names

Considerable care should be exercised in devising names for data, paragraphs, and sections to make them as helpful as possible to the reader in understanding the function and structure of the parts of the program. Therefore, installations should consider adopting conventions that encourage the use of meaningful names. An example of such a convention would be to prefix the names of transaction file items with T, old master file items with M, and new master file items with NM. Another possible convention, not consistent with the first, is that data name qualification be used consistently to convey information about data organization. Another would be that file names must contain the word FILE and record names the word RECORD or perhaps REC. A possible convention for paragraph naming would be to require that all paragraph names state the paragraph function and begin with numbers denoting sequence or hierarchy in the program. Many other such conventions are possible. Once learned, their use involves very little extra effort.

Comments

Experience has shown that well-structured COBOL code can be largely self-documenting, assuming the use of descriptive data and paragraph names It is recommended that an attempt be made to write programs without comments. If comments are used, however, they should be entered in the form of comment lines with an asterisk in column 7. The NOTE statement should be avoided, since minor errors in writing it frequently lead to programs that cannot be correctly compiled. If it is found necessary to use comments, they should be organized and formatted so that they do not interfere with the readability of the program. Free use of blank lines will make comments stand out from associated code.

The ALTER Statement

Many experienced COBOL programmers suggest that use of the ALTER statement in connection with the GO TO is so difficult to document adequately, and so likely to lead to serious difficulties in program test and maintenance, that the ALTER should never be used under any circumstances. To accomplish the same purpose, there are alternative ways which are easier to understand and simpler to maintain.

Special Conditions

Most programming environments allow for specified unusual conditions to interrupt the normal flow of processing and activate exception-handling routines. Common examples are end-of-file conditions and arithmetic overflow. Whether the structure theorem applies to programs containing such elements depends on whether they violate the one-entry, one-exit principle and thus fail to be proper programs. Certain types of interrupts always break the normal flow; others may or may not, depending on how the program is written.

The COBOL Declaratives Section can be used to specify the desired processing for certain events that cannot normally be tested for by the programmer. Examples include label processing, input/output error routines, and certain functions of the report writer. The Declaratives Section, if present, must appear at the start of the Procedure Division; it cannot be entered other than by the occurrence of the event that the code is designed to handle. Since these blocks of code are out of line, they involve a transfer of control, even if it is transparent to the programmer. Such interruptions of sequential control are usually considered undesirable in structured programming. However, because of the utility of the Declaratives Section and the difficulty or impossibility of carrying out the functions any other way in COBOL, no attempt has been made to restrict usage of the feature. The problem is lessened somewhat by the operation of the interrupt in COBOL: control automatically returns to the statement following the one that caused the interrupt.

A conditional statement in COBOL contains a condition that is tested within a phrase to determine which of the alternate paths of program flow is to be taken. Examples are the READ which always has either an AT END phrase or an INVALID KEY phrase; arithmetic verbs which may include an ON SIZE ERROR phrase; the SEARCH verb which contains one or more WHEN phrases and may contain an AT END phrase. In each case it is permissible to write one or more imperative statements which are executed if the associated condition is true. For example, there may be a PERFORM statement used within the set of imperative statements, or NEXT SENTENCE may be used. In almost every case, control should be given to the imperative statement following the conditional statement.

Occasionally, it is not feasible to handle certain conditions within a series of statements. This situation may arise either within conditional statements or during normal processing, for example, when errors are detected in data editing which prevent further processing. The programmer has at least two methods of handling such situations. One is to set a flag and then return control to the next-higher level routine for further action. This technique usually works well, and there are no violations of structured programming conventions. If, however, the error is detected within the innermost level of many nested levels, many tests may be required (one at each level) to return control up through the nested structures to the point where the error can be handled. Another alternative is to allow the use of GO TO for such disabling error routines. Good judgment should be used to determine whether the maintainability of the program is improved by using a few GO TO's in this case.

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

AUTOR:	KENNETH T. ORR
TITULO:	STRUCTURED SYSTEMS DEVELOPMENT
EDITORIAL :	YOURDON PRESS
AÑO:	1977

CAPITULO 2:	EL MODELO
CAPITULO 3:	ANALISIS, SINTESIS Y FLUJO DE PROCESOS

EL MODELO

En el Diseño de Sistemas Estructurados, nos aprovechamos de un número de herramientas y conceptos que han sido desarrollados específicamente para ayudarnos a pensar acerca del manejo de información. Las más de estas herramientas son simples de entender, pero ellas no son siempre simples de usar, al menos no, inicialmente.

A medida que llegue a entender la Teoría que está detrás, usted encontrará su aplicación más natural.

El Diseño de Sistemas Estructurados es un enfoque basado sobre un Modelo de Sistemas más bien simple. Aquél modelo se fundamenta en el hecho de que cualquier Sistema (de Información), puede ser considerado como constituído de tres partes básicas :

1. Salida.
2. Una Caja Negra que produce la Salida y,
3. Entrada de la cual la Caja Negra produce la Salida. (Esto podría verse como una manera peculiar de decir que un Sistema está hecho Entradas, una Caja Negra y Salidas, sin embargo, usted verá el punto más tarde). Una versión gráfica del concepto se señala a –

continuación :

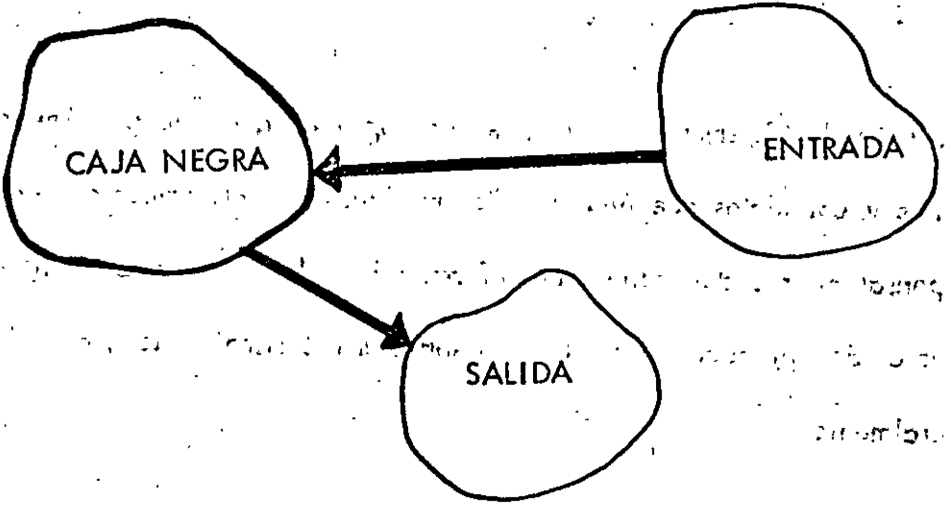


FIG. 2.1. Diseño de Sistemas Estructurados, basado en un Modelo de Tres - Partes.

Al desarrollar un Sistema, el Analista de Sistemas necesita un Modelo Mental. Este tiene ciertas ventajas :

1. Es Simple.
2. Es tan claro como muchos de los Sistemas, que usted está apto para encontrar en el mundo real y,
3. Lleva a cosas más grandes y mejores. Pero antes de hablar acerca de aquéllas cosas más grandes y mejores, necesitamos refinar nuestro Modelo un poquito.

Suponga que empezamos por listar las cosas que sabemos debemos tener, o que se ha dicho que necesitamos, en un Sistema.

Primero, necesitaremos Salidas, dado que ellas son la razón principal de desarrollar cualquier Sistema; y para producir Salidas, probablemente necesitaremos Entradas. Pero ¿ qué queremos decir exactamente con Salidas y Entradas ?, - por ejemplo, ¿ queremos decir reportes ó archivos, o piezas de papel o bits sobre una línea telefónica?. En general, por Salidas queremos decir solamente aquéllas cosas con las cuales el cliente del Sistema trata actualmente.

Ejemplo :

> Una Salida puede ser un Reporte, una Pantalla sobre una Terminal - de Video, o una Respuesta Auditiva sobre un Teléfono. Una Entrada puede significar una forma de transacción, un cheque sin fondos o -- una llamada telefónica.

En general, ni el término quiere decir bits electrónicos sobre una Cinta o Disco, o un reporte de Auditoría Interna. (Estas cosas son importantes, pero ellas representan datos internos - diremos más acerca de ellas más tarde).

En el Diseño de Sistemas estructurados, las Salidas y Entradas (junto con ciertos aspectos de la Caja Negra), caen dentro de una categoría llamada <u>Requisistos del Sistema.</u> Mientras tratamos con un número de elementos, <u>los Requisitos del Sistema están relacionados primariamente con las Salidas y las reglas lógicas para su derivación.</u>

Algo sorpresivamente, desde el punto de vista de los requisitos, estamos mucho - menos relacionados con Entradas que con Salidas.

La razón de esto, llegará a ser más clara posteriormente.

Si un cambio es hecho a un Sistema y si afecta la Salida, entonces es un cambio a los requisitos del Sistema. Si por otra parte, usted convierte un Sistema de un Sistema Operativo de Computadora a otro, o si cambia la secuencia de operaciones, sin afectar la Salida producida por el Sistema, entonces aquél cambio está envuelto con la Arquitectura de los Sistemas.

La Arquitectura de los Sistemas está comprometida con el trayecto de Salida, - que es producido en la Entrada.

Mientras los Requisitos de los Sistemas primariamente envuelven Salidas y sus derivaciones, la Arquitectura del Sistema está en gran medida relacionada con la composición de la Caja Negra, esto es, la organización de las piezas del Sistema. La Información es registrada, procesada, almacenada y publicada. La Arquitectura del Sistema está relacionada como este proceso producirá los Resultados correctos.

Cuando empezamos a desarrollar un sistema (basada sobre nuestra experiencia - pasada), hacemos frente a algunos problemas serios de entender que vamos a - construír. Cada uno de nosotros lleva en su cabeza un Modelo de lo que vamos a producir; y verdaderamente, si no lo llevamos, nunca realizaríamos nada.

Sin embargo, si nuestro Modelo es erróneo, o si nos lleva a hacer el tipo de cosas erróneas, aquél Modelo Mental puede hacer un, de por sí trabajo difícil en imposible.

Así, nuestro Modelo está basado sobre las observaciones de que el rasgo único más importante de un Sistema es su Salida, y que el Camino más Fácil de determinar la efectividad de un Sistema, es si este siempre produce la Salida correcta deseada. Si no es así, usted no habrá tenido éxito en construír un Sistema correcto, no importa que tan rápido corra, o que Entradas acepte.

Miremos un Sistema desde otro punto de vista ligeramente diferente. Las Salidas y Entradas pueden ser pensadas como " Conjunto de Datos ". Por ejemplo, un Reporte de Nómina puede ser considerado como un conjunto de datos acerca de los empleados. Por lo tanto, la Caja Negra en nuestro Modelo contiene conjuntos de datos; pero también contiene otra cosa: " Conjunto de Acciones ", (u operaciones). Entonces, un Sistema correcto es uno en el cual - los conjuntos correctos de acciones son ejecutados sobre los conjuntos de datos correctos en tiempos correctos.

Si el Conjunto correcto de acciones mensuales (llamados Programas de Reporte), son hechos sobre los conjuntos de datos correctos, (llamados Archivos -- de Nómina), en el tiempo correcto (llamado Contabilidad de Fín de Mes), -

entonces usted producirá la Salida correcta y tendrá un Sistema correcto. Un Sistema correcto es el criterio mínimo para hacer trabajo de Sistemas. Sin embargo, además de producir la Salida correcta, usted tipicamente también tendrá que producir aquélla Salida a un cierto costo y dentro de un esquema de tiempo dado, prescrito por el gerente. Pero primero, usted debe producir las Salidas correctas.

Pero ¿ cuáles son las Salidas correctas ?, pues bien, identificarlas es la mitad del trabajo del Analista de Sistemas. Esto es por lo que, en nuestro Modelo original, los Esquemas para la Salida, la Caja Negra y las Entradas, fueron tan borrosos.

En muchos casos, nosotros simplemente no conocemos al principio, cual es la respuesta correcta. Un gerente puede conocer que él o ella necesita tener información más oportuna o necesita proteger el inventario pero no con mucho detalle.

Esto no significa que usted nunca convencerá a un cliente a ser específico acerca de lo que el quiere. En efecto, algunos clientes pueden ser bastante específicos. Pero usted debe entender que en la mayoría de los casos, usted tendrá que ayudar al cliente a pensar acerca de su problema y presentarse con un Sistema que le ayude a hacer un mejor trabajo.

En general, al desarrollar un sistema, mi trabajo con clientes ha sido un proceso de refinamiento contínuo (y redefinición) del problema, para producir una - solución. Muchas veces lo que el usuario dice es el problema que viene a resultar sólo un síntoma.

Como buen doctor, un buen analista de sistemas debe tener un punto de vista - más allá de los síntomas para tratar la enfermedad detrás.

Usualmente, en los primeros pasos de los proyectos de Sistemas, solo unos cuantos requisitos fundamentales de Sistemas son absolutamente ciertos, a menos, por supuesto, que usted haya hecho exactamente el mísmo sistema antes, y aún así, usted tiene que ser cuidadoso. En un sistema de nómina, por ejemplo (ver - FIG. 2.2.), sabemos que tenemos que producir cheques de pago y ciertos -- Reportes para el Gobierno.

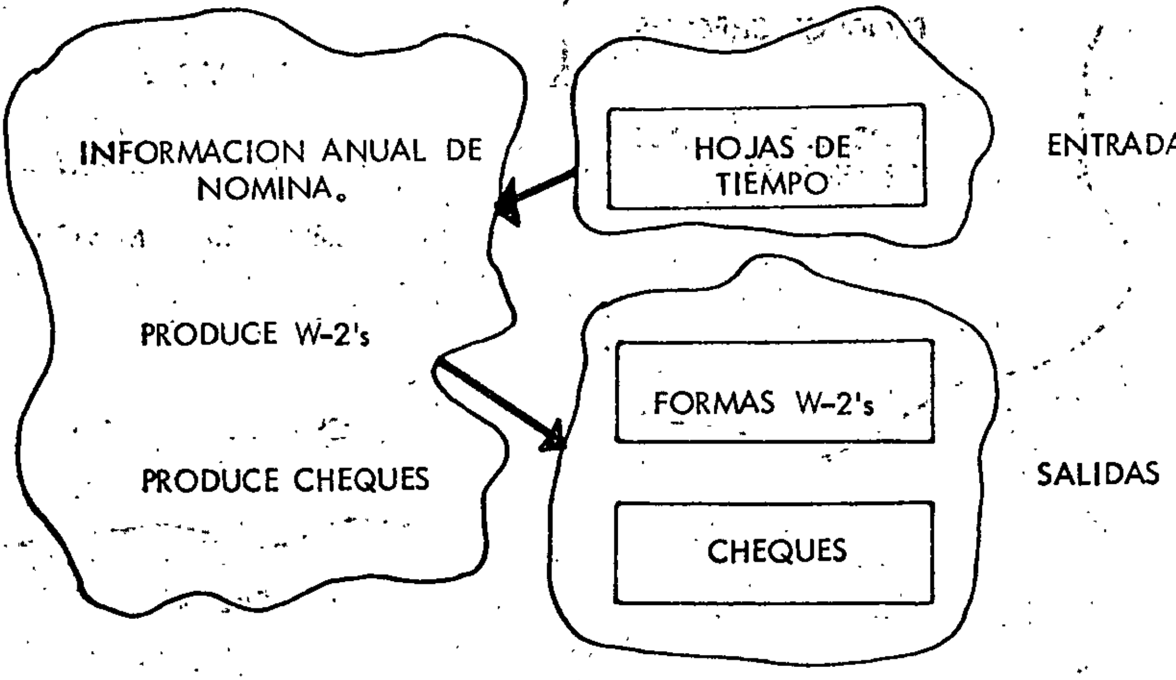


FIG. 2.2. Requisitos fundamentales de los Sistemas.

A medida que aprendamos más acerca de lo que estamos tratando de sistematizar, encontraremos más reportes que son obligatorios, en otras palabras, Reportes para el Gobiemo Local y Estatal o Reportes de Cuentas Básicas, solicitados por el Departamento de Cuentas de la Firma C.P.A., mientras pueda haber muchos de tales requisitos, tal vez, ellos son la parte más fácil de un Sistema con los cuales tratar; porque ellos están usualmente bien definidos. El usuario conoce lo que ellos deben contener y como ellos se deben mirar. En esta etapa nuestro sistema está empezando a tomar forma.

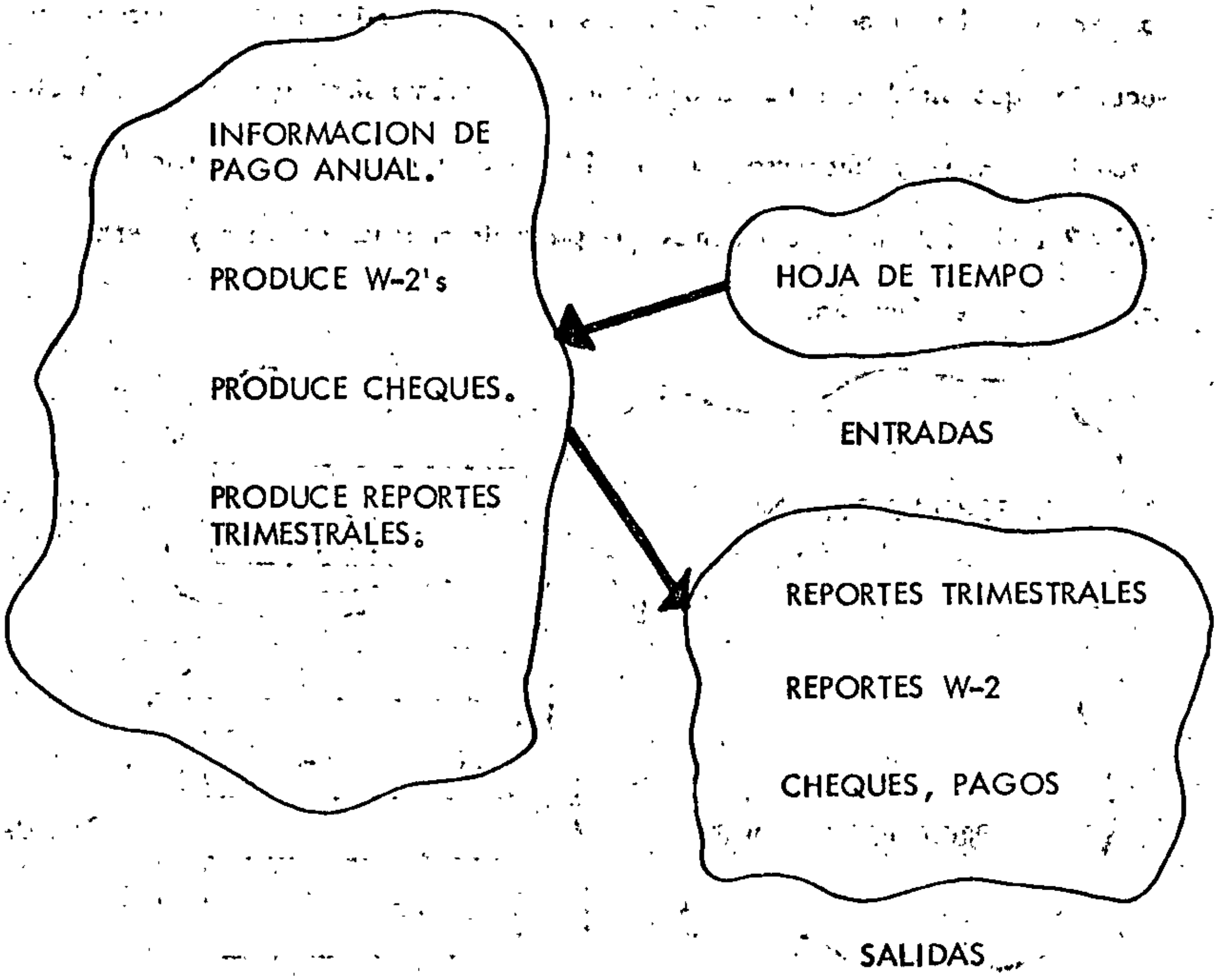


FIG. 2.3. Requisitos Adicionales de los Sistemas.

En cada paso de refinamiento, aprendemos más acerca de los conjuntos de datos y conjuntos de acciones de los sistemas. Intelectualmente, para tratar con todos estos datos, el analista tiene que sumarizar o clasificar esta información en estructuras que puedan ser manipuladas fácilmente. Por ejemplo, el analista debe empezar a hablar acerca de reportes semanales o de reportes mensuales como un grupo. O el puede categorizarlas como obligatorias u opcionales, o federales o de la compañía.

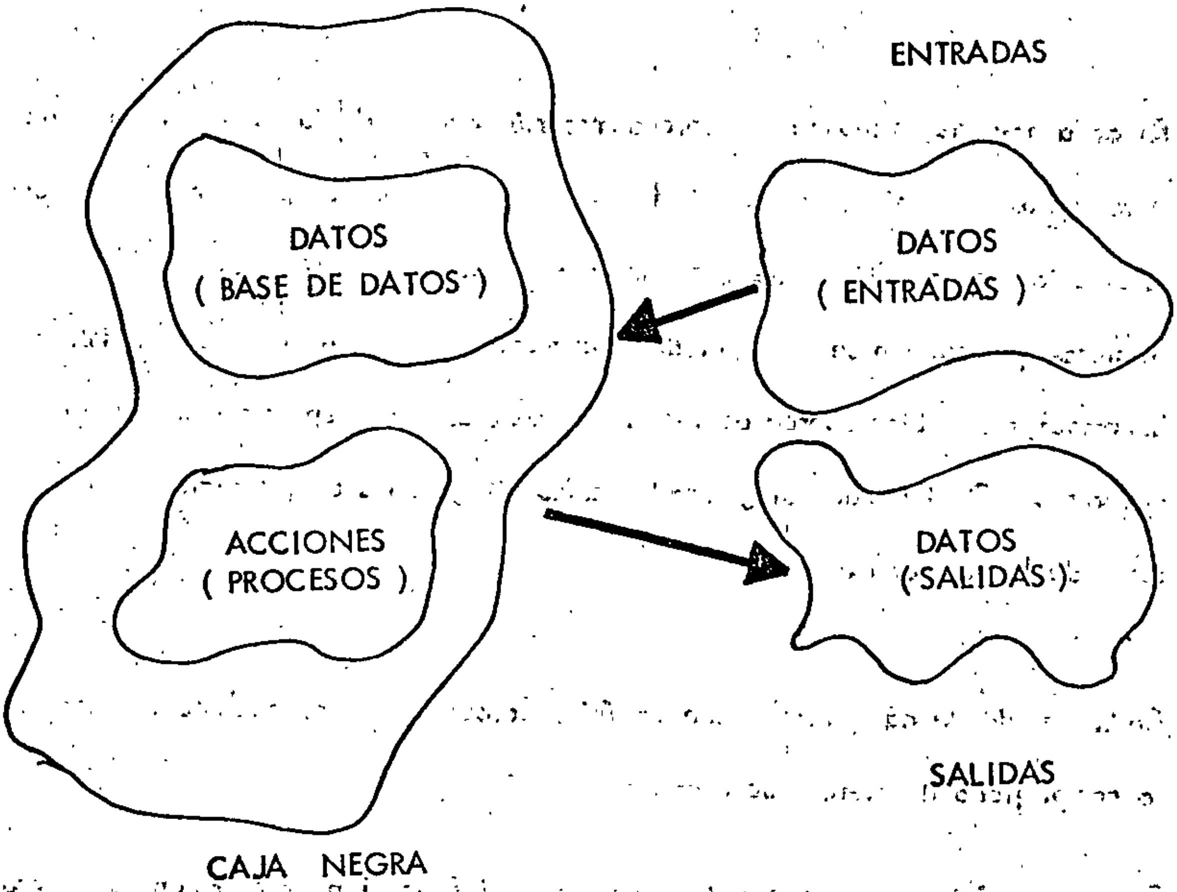
Cada método de agruparlas puede ser útil, porque ayuda al analista a tratar con la complejidad de cientos de detalles.

En efecto, sin alguna manera de organizar la información para encontrar el gran esquema, es altamente improbable que usted pueda partir. Muchos esfuerzos en sistemas fallan simplemente porque el analista se hundió en un mar de detalles.

Debemos esforzarnos por evitar esta falla. Otra vez, nuestro modelo puede ayudar.

Dijimos anteriormente que la parte de la Caja Negra de nuestro Sistema, puede ser considerado como conjuntos de acciones o conjuntos de datos.

Mostraremos eso gráficamente a continuación :



Este refinamiento de nuestro modelo de sistemas, tiene agregado una estructura adicional, para éste, muestra como las piezas básicas actualmente se relacionan con otra. Ahora, podemos pensar en un sistema en términos de la relación entre conjuntos de acciones y conjuntos de datos.

Tambiém podemos dar nombres a los elementos principales :

- Salida.
- Proceso.
- Base de Datos.
- Entrada.

Conceptualmente, nuestro sistema parece que se ajusta a nuestras ideas del mundo real. Primero, coleccionaremos información (Entradas); entonces procesamos la información (Proceso), colocamos algo de la información procesada en forma de reportes (Salidas); y algo de ella la almacenamos para uso futuro -- (Base de Datos). Pero tenemos que ser cuidadosos porque la manera más natural de pensar acerca de un problema envuelve Entrada, Proceso, Salida.

Ciertamente , ese es el camino con el cual estamos aptos para procesar los datos cuando el sistema se este construyendo, pero no es el orden en el cual - consideraremos las cosas cuando diseñemos un sistema.

Para diseñar un sistema, partimos con la Salida y trabajamos hacia atrás por - todo el sistema, hasta que usted finalmente llegue a los pasos de procesamiento que son requeridos, la base de datos que usted necesita y finalmente las Entradas que usted tiene que colectar. A primera vista, en sistemas el trabajo - hacia atrás se ve no - natural. La razón es simplemente que partimos por el - principio, i.e., con las Entradas, y vamos hacia adelante no trabajando muy - bien.

Es como tratar de ir en un viaje sin primero haber decidido donde vamos a ir. ¿ Cómo sabemos que empacar ?, dado que usted no conoce donde va a ir, usted no conoce el clima que habrá, y por lo tanto, usted probablemente empacará mucha ropa, justo para cada caso.

Su siguiente problema se encuentra en el momento que usted abandone la carretera. ¿ Cuál camino debe tomar ? .

La importancia de tener metas u objetivos es claro, en cualquier cosa que - hagamos. Pero porque, si toda persona paga jarabe de pico para planificar, encontrar metas y así por el estilo, nosotros a menudo remendamos chapuceramente nuestro trabajo de sistemas.

Una razón es porque simplemente nosotros no estamos apropiada y suficiente-- mente orientados - a - Salida. Más tarde, veremos que partiendo en la Entrada, en lugar de Salida, es a menudo por lo que fallamos para completar - exitosamente muchos esfuerzos en Sistemas. El Diseño de Sistemas por las Entradas, se asemeja en mucho cuando al pintar se deja usted mismo encerrado - en una esquina. Piense en ésto : ...Está en el lugar correcto cuando termina de pintar un piso (i.e. en la puerta), usted tiene que trabajar hacia atrás de la puerta, a la posición de Salida, donde usted estuvo al partir. No hay nada complicado acerca de este enfoque, pero es más duro de aplicar al diseño de Sistemas de lo que usted podría pensar.

Un número de beneficios resultan de trabajar hacia atrás. Un beneficio principal es que en el caso ideal (lo cual probablemente nunca sucederá), us-- ted no coleccionará o almacenará datos que usted no use - un factor cierto - que tiene un efecto principal en el costo y la complejidad del producto terminado - ninguna mención sobre preguntas de seguridad de datos y privacidad.

Otros beneficios serán discutidos, así como sean descubiertos.

El Diseño de Sistemas Estructurados, ayuda a producir Sistemas Mínimos. En esta etapa de Sistemas Flexibles, Generales y Poderosos, queremos diseñar algo que sea mínimo.

¿ Qué hace a un Sistema Mínimo ?, ¿ Cómo lo definirías ?.
Un Sistema Mínimo hace el mínimo absoluto para producir las Salidas deseadas. Tal Sistema sólo captura datos que aparecen como Salidas o que afecta algo que llega a salir.

Otro camino de mirar nuestra definición es pensar en usted mismo como un Matemático que ha derivado una fórmula perfecta para producir un Diseño de Sistemas correcto cada vez. En tal fórmula matemática, si se da un conjunto de valores para las variables independientes, entonces todas las variables dependientes pueden ser mecánicamente computadas. ¿ Pero cuáles son exactamente llas variables independientes en el Diseño de Sistemas ? . Ellas serían aquélos elementos que determinan cualquier cosa ¿ No es verdad ?, ¿ Qué categoría de cosas con las que tratamos en el trabajo de Sistemas se ajustan a ésa descripción ?. Por supuesto, las Salidas.

Si cambiamos los requisitos de Salida, cualquier otra cosa tiene que cambiar en el Diseño de Sistemas para acomodarlo. Si la Salida cambia mínimamente, debemos cambiar el proceso y si nosotros no tenemos los datos del cual la nueva Salida se deriva, entonces el cambio afectará la Base de Datos y también las

Entradas. Por lo tanto, los requisitos de Salida pueden ser pensados como - variables independientes del proceso de sistemas.

Nuestro modelo tiene cuatro partes fundamentales, algunas de las cuales son más importantes que otras. En general, queremos concentrarnos sobre las Salidas y entonces trabajar nuestro camino hacia atrás, para determinar que - proceso, que base de datos y que entradas serán necesarios para producir - aquéllas Salidas. Hasta aquí todo está bien. Tenemos un modelo, y conocemos con cual fin del problema partir.

En teoría, trabajar hacia atrás de la Salida, se ve razonable; y lo más que pensará acerca de él, de lo más que usted se dará cuenta, es que es el único enfoque razonable. Sin embargo, en la práctica trabajar hacia atrás, - es difícil de aplicar.

Una de las razones principales es que pensar en Salidas es un trabajo duro. Requiere que usted establezca metas y objetivos y que usted haga decisiones fuertes tales como : ¿ Porqué necesitamos esta información ?, ¿ Cómo quiero ordenar mi información ?. Por otra parte, es mucho más fácil hablar de Entradas, las cuestiones difíciles siempre pueden ser dejadas para más tarde; usted puede simplemente decir que no tiene que ser demasiado específico - acerca de como usará la información.

Debemos coleccionarlo porque algún día puede ser necesitado.

La mayoría de los enfoques del pasado han indicado que una de las primeras tareas de Sistemas (justo después de especificar las Salidas, es definir las - Entradas con el usuario. Existe un número de presiones fuertes para considerar la definición de Entradas fuera de secuencia.

En el siguiente Capítulo, discutiremos los conceptos de Análisis y Síntesis como herramientas en el proceso de construcción de Sistemas; y después de describir algunas de las herramientas que pueden ser usadas en el diseño de Sistemas Estructurados, retornaremos a nuestro modelo de un Sistema y empezaremos a ver como aplicar lo que ya conocemos para construír Sistemas.

Hasta aquí, hemos hablado casi en generalidades, pero estamos asentando la infraestructura para otras cosas.
En particular, estamos intentando ayudarlo, al analista, viendo como un Sistema crece de la idea a una operación tangible. El Diseño es un trabajo difícil, - pero divertido; sin embargo, facilmente puede salirse de la mano.

También a menudo, promovemos gente en responsabilidades de Análisis de Sistemas, sin enseñarles como hacer análisis o Diseño de Sistemas.

Esto puede resultar en un error principal. Como con cualquier actividad hay caminos correctos para atacar el diseño e implementación de un Sistema y hay caminos incorrectos. Aprendemos que muchas de las guías antiguas acerca de la construcción de Sistemas son en realidad sólo Mitos.

Por algún tiempo, hemos necesitado un nuevo modelo mejorado. Ahora que tenemos uno, empezaremos a ver los resultados de tener un modelo mejor con el cual trabajar.

17

ANALISIS, SINTESIS Y FLUJO DE PROCESOS.

En el último Capítulo, empezamos a desarrollar un modelo intelectual para un sistema de información. Nosotros retornaremos a aquél modelo después que hayamos mirado como algunos enfoques tradicionales son usados para analizar y diseñar sistemas. El propósito de introducir enfoques tradicionales en este punto es la de darnos prácticas en el trabajo de sistemas antes de que debamos usar nuestro conocimiento en un ambiente estructurado.

El modelo del último capítulo sirvió a una función muy importante: Nos dió un armazón conceptual o teoría.

Nosotros recogeremos más teoría, así como prosigamos.

Muchos de nosotros vamos por la vida entera, sin reconocer la importancia de la teoría en cualquier cosa que hacemos, primariamente porque las teorías se ven demasiado abstractas y remotas para ser de mucho uso. Las personas en profesiones nuevas a menudo están demasiado ocupadas haciendo reflejar lo que ellos están haciendo o como se relacionan con otras cosas.

Tomemos la programación de las computadoras como un ejemplo. Por años, las personas desarrollaron programa tras programa, usando una variedad de máquinas y lenguajes, convencidas que la programación era, a lo más, un arte.

La mayoría de los programadores sentían que mientras aprendían a programar, no había caminos científicos para escribir un programa correcto. Un buen - programa era simplemente un programa que trabajaba, y el único camino de- conocer en la actualidad si un programa trabajaba, era probar y probar y probar. Aún entonces, usted no podría estar absolutamente seguro si siempre -- trabajaría.

Hoy, sabemos que existen caminos mejores de desarrollar programas correctos. Conocemos, por ejemplo, que si lógicamente entendemos lo que queremos hacer, podemos construír un programa correcto para hacerlo. Por otra parte, - estamos bien seguros que podemos hacer que nuestros programas trabajen, si - no la primera vez, al menos en unos cuantos ensayos. ¿ Cómo ha surgido - esta transición de la duda a la certeza en la programación en tan corto --- tiempo ?.

La respuesta simple es que ahora tenemos una teoría para guiar nuestras actividades - una teoría para programar. Ahora conocemos que si resolvemos todas las posibilidades lógicas en detalle y construímos nuestro programa de una manera más lógica jerárquica, trabajará.

También conocemos que podemos usar la estructura de los datos como una guía para la construcción de nuestros programas, eliminando de tal modo muchas -

./.

decisiones difíciles, una vez que nos enfrentamos con la organización y estructura de nuestros programas.

Si aplicamos esta misma técnica de análisis lógico jerárquico a un sistema total, incrementalmente podemos llegar a estar seguros que desarrollaremos un sistema correcto, i. e., uno que realice el conjunto correcto de acciones sobre los conjuntos correctos de datos en tiempos correctos. Sin embargo, hay un número de diferencias entre el diseño y construcción de un programa y el diseño y construcción de un sistema completo.

Por un lado, el programador ordinario tiene menos preocupaciones que las que tiene el analista de sistemas. A menudo un analista o cualquier otra persona ya han definido las salidas, las entradas y el proceso básico para el programador. Así, mientras que el programador tiene que descifrar muchos detalles, el no tiene que definir parámetros básicos.

Por otra parte, el analista está en una situación diferente: El no sólo tiene que descifrar que necesita tener que hacer, el tiene que hacerlo bien. Ninguna otra cosa excepto el problema le es dado al analista. El tiene que sumar sus propias ideas a los deseos de las varias personas que deben interactuar con el sistema para definir la extensión del problema. Esto llega a ser el porqué y el qué del sistema.

./.

Entonces el tiene que descifrar un camino por el cual hacer estas cosas - el cómo y cuándo del sistema.

Resolver esta relación es integral al proceso del diseño. Claramente, la determinación del porqué y qué y del cómo y cuándo están relacionadas. En un - sentido clásico, estamos hablando acerca de dos actividades: análisis y síntesis. Por lo tanto, miraremos el análisis y la síntesis de sistemas (también -- llamadas diseño o integración), y veremos como se relacionan al proceso real de diseñar un sistema.

Análisis : Resolución dentro de elementos simples.

- Webster's Unabridged Dictionary.

Análisis es una palabra antigua que significa romper algo en sus piezas fundamentales. Esa definición es verdadera, ya sea que se refiera al análisis químico, al análisis linguistico o al análisis de sistemas. En el análisis de sistemas, estamos relacionadas con romper un sistema en piezas y con las herramientas -- que nos sirven para hacerlo.

En general, existe un enfoque jerárquico característico de todos los tipos de -- análisis; esto es, romper algo grande en piezas sucesivamente más pequeñas, -- hasta que no exista posibilidad de subdividir posteriormente cualquiera de las -

./.

piezas. Cada nueva pieza está subordinada a la pieza anterior.

¿ Cómo trabaja en la actualidad el proceso del análisis jerárquico ?, tomemos un ejemplo: suponga que tiene asignado el problema de desarrollar un - sistema para producir un reporte estadístico mensual, como es representado -- abajo:

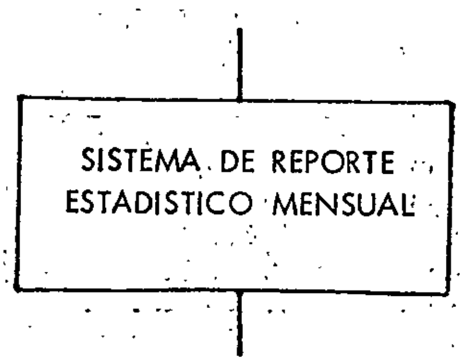


Figura 3.1. Problema para producir reportes mensuales.

En el análisis jerárquico preguntamos, ¿ Cómo podemos separar esta tarea en una serie de partes más simples ?. Un camino es pensar en algunas de las - partes básicas del problema que podía creeerse se encuentran en el problema de la definición original. En este caso, nuestra meta es desarrollar un sistema que produzca un reporte estadístico mensual.

De experiencia pasada, podríamos esperar (al menos del lado de la computadora) producir aquél reporte mensual en un número de pasos: Por ejemplo, - una parte del sistema produciría el reporte, otra parte podría actualizar el - archivo maestro (ó base de datos), del cual los reportes serán producidos —

./.

y otra más podría inicialmente revisar los datos de entrada para excluír cualquier valor erróneo. El método típico de mostrar esta relación es dibujando - algo parecido a un diagrama organizacional :

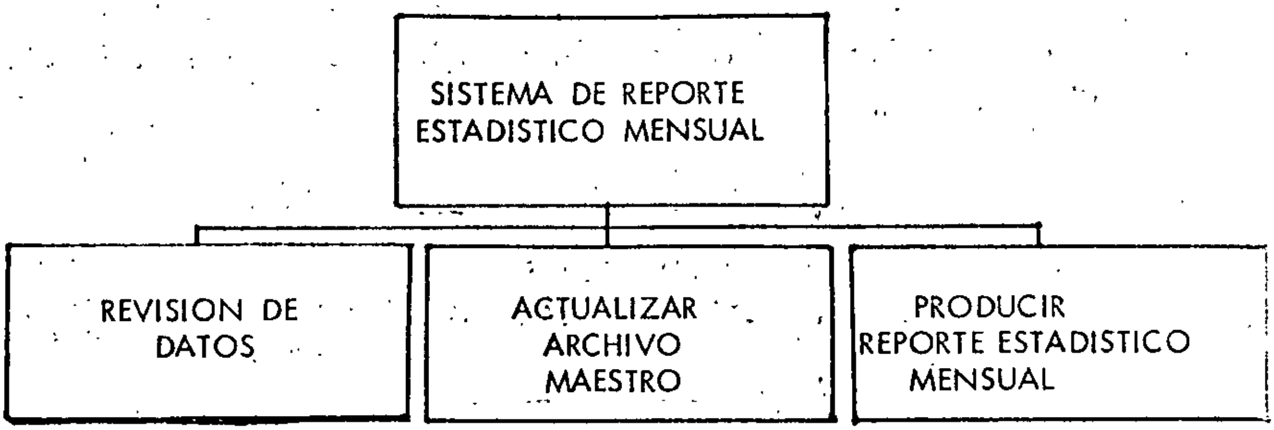

SISTEMA DE REPORTE ESTADISTICO MENSUAL
REVISION DE DATOS
ACTUALIZAR ARCHIVO MAESTRO
PRODUCIR REPORTE ESTADISTICO MENSUAL

Figura 3.2. El análisis jerárquico requiere romper el problema.

Un análisis jerárquico similar se maneja en casi cualquier tarea de sistemas y, en efecto, es tan común y natural que muchos analistas no se molestan en -- escribirlo. Ellos simplemente asumen que cualquiera entiende como se llegó - a su diseño. A menudo, este no es el caso y, es importante para aquéllos -- que implementarán, mantendrán u operarán el sistema entender también como - fue construído.

El análisis de un problema en partes es un proceso útil y es más efectivo si - sistemáticamente se continúa aplicando el mismo proceso de dividir y vencer -

./.

cada parte del probl ema.

Por ejemplo, podríamos subdividir (o analizar), la parte llamada PRODUCIR REPORTE ESTADISTICO MENSUAL en varias partes: una para leer el archivo maestro, una para sumarizar los datos y una para diseñar e imprimir el reporte como sigue :

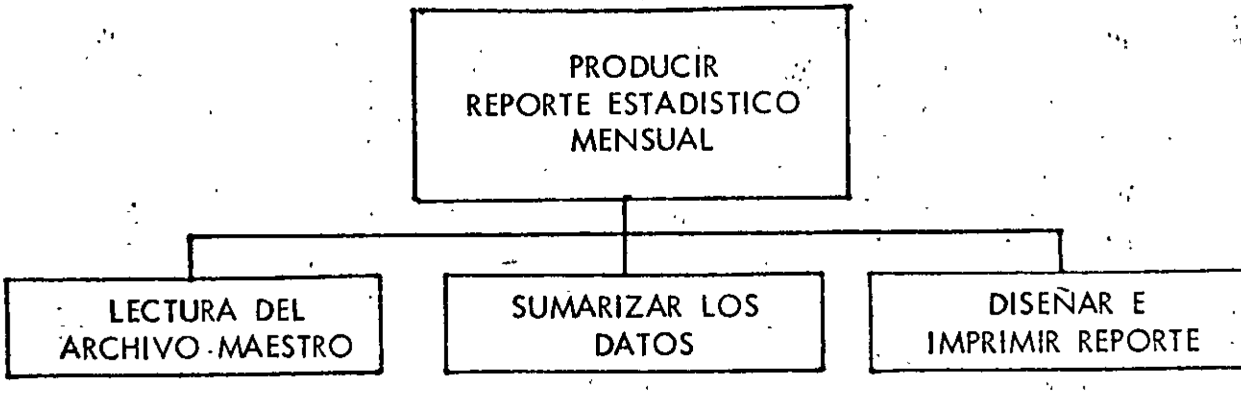
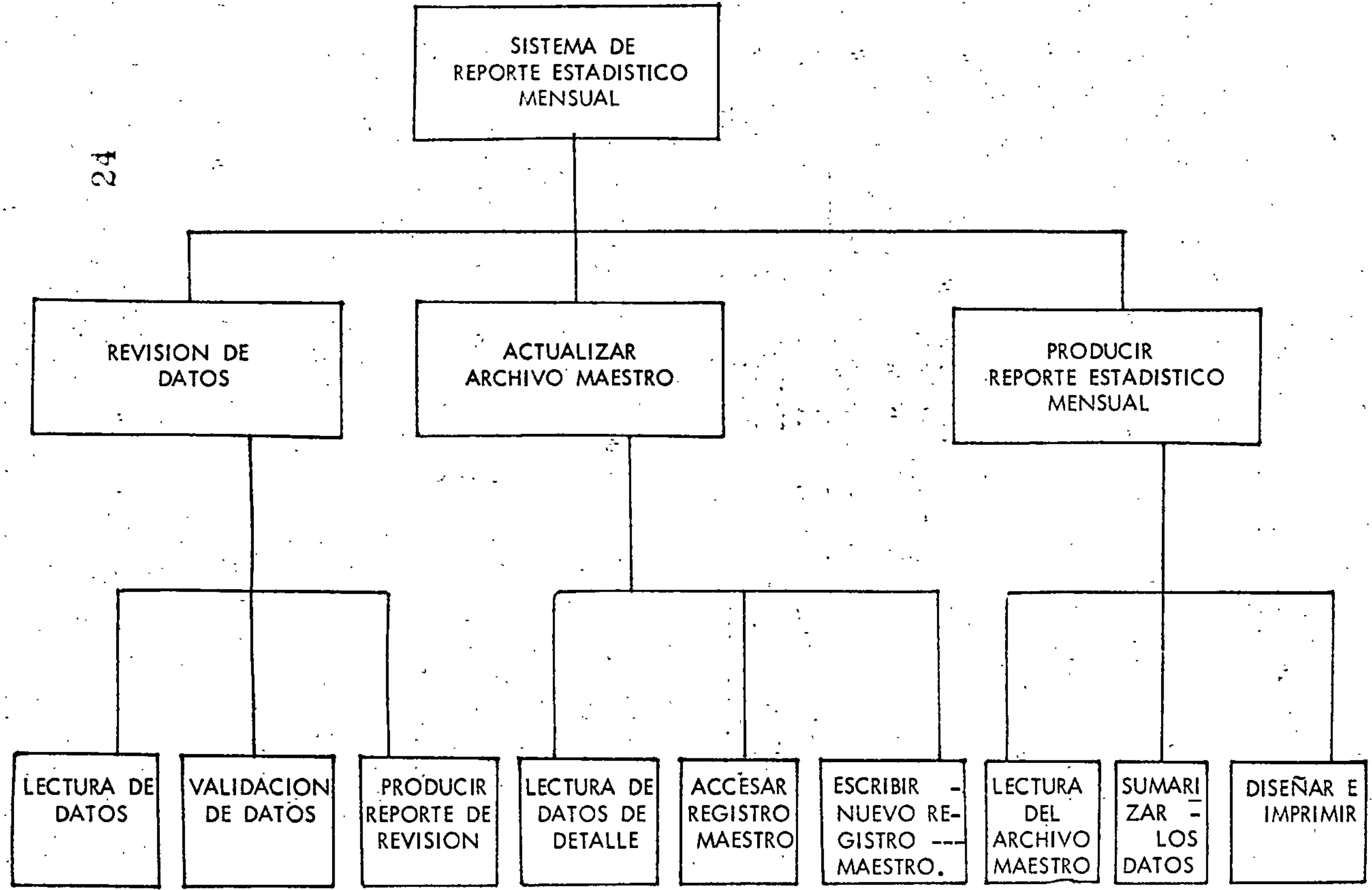


Figura 3.3. Ruptura del problema en pasos más detallados.

Verdaderamente, podríamos hacer lo mismo con todas las demás piezas del - rompecabezas :

./.


SISTEMA DE
REPORTE ESTADISTICO
MENSUAL
REVISION DE
DATOS
ACTUALIZAR
ARCHIVO MAESTRO
PRODUCIR
REPORTE ESTADISTICO
MENSUAL
LECTURA DE
DATOS
VALIDACION
DE DATOS
PRODUCIR
REPORTE DE
REVISION
LECTURA DE
DATOS DE
DETALLE
ACCESAR
REGISTRO
MAESTRO
ESCRIBIR -
NUEVO RE-
GISTRO ---
MAESTRO.
LECTURA
DEL
ARCHIVO
MAESTRO
SUMARI
ZAR -
LOS
DATOS
DISEÑAR E
IMPRIMIR

En un análisis completo, este proceso continúa hasta que cada una de las piezas es tan simple que no hay necesidad de hacer divisiones posteriores.

Si se realiza sistemáticamente, el análisis es una herramienta extremadamente poderosa. Cuando termine, usted tendrá una idea completa de que necesita tener que hacer y como hacerlo.

Conceptualmente, el análisis es una técnica simple que trabajará todo el tiempo. Através del tiempo, casi cualquier analista de sistemas de éxito ha empleado una u otra forma del análisis jerárquico.

Actualmente, el análisis está disfrutando nueva popularidad bajo el nombre de diseño de arriba - a - abajo (top-down), el cual recomienda que cada problema sea analizado paso por paso, hasta que esté completamente resuelto.

La metodología HIPO (Hierarchical-Input-Process-Output), utiliza muy extensivamente el análisis jerárquico.

Si el análisis jerárquico es tan natural y tan popular, usted podría asombrarse porque no lo hemos usado con mayor frecuencia. Existe un número de razones una es que se ve como una gran cantidad de trabajo, especialmente en los niveles más bajos. Por otra parte, requiere que usted tenga alguna idea de la

./.

cima del sistema y de las piezas principales en cada paso. Finalmente, el análisis jerárquico plantea la cuestión: En primer lugar, ¿ Cómo conocemos en que piezas descomponer el Sistema ? .

Este problema es difícil. El ejemplo anterior del sistema de reporte mensual no era complicado, pero suponga que tenemos que construír un sistema completo de información a la gerencia. Ciertamente, el problema sería más -- complejo, y el análisis de arriba-a-abajo, normalmente sería muy dificil de realizar. Por esa razón, el análisis jerárquico no había sido reconocido como una herramienta fundamental en el trabajo de sistemas, sólo hasta muy - recientemente a consecuencia de su uso común.

El análisis tiene un importante producto: hace que los sistemas sean jerárquicos y modulares. Esto es extremadamente importante, nuestra experiencia -- sugiere que todos los sistemas buenos son jerárquicos y modulares.

Sin embargo, el análisis no trata con una cuestión fundamental del diseño: - ¿ Cómo se relacionan las piezas (módulos), de la jerarquía una con otra ? Esta cuestión concierne principalmente a la síntesis.

Sin el análisis está relacionado con descomponer un problema en partes, entonces, la síntesis está relacionada con colocar juntas las piezas de un sistema .

./.

En efecto, un término que se usa a menudo para significar síntesis es "integración", la síntesis está relacionada con el orden y con el tiempo.

Si el análisis está relacionado con el qué y el porqué, entonces la síntesis está relacionada con el cómo y cuándo.

En el diseño vemos dos fases: descomponer y ajustar juntas. En nuestro ejemplo, el sistema se descompuso en un número de partes: una que revisaba las transcacciones, otra que actualizaba el archivo maestro y otra que producía el reporte mensual. Pero nosotros podemos arreglar aquéllas piezas derivadas del paso del análisis en un número de caminos diferentes. Por ejemplo, los dos sistemas de la figura 3.5. tienen las mismas piezas.

El analista de sistemas experimentado reconocerá que la figura 3.5.b. es una solución más realista que la 3.5.a. Aunque las piezas principales son las - mismas, la cuestión del cuándo y cómo, han sido consideradas con mayor detalle.

También, una pieza adicional ha sido descubierta y quefué pasada por alto - en nuestro primer análisis (o incluída en nuestra definición de REVISION DE DATOS). Al ajustar juntas las piezas otra vez, a menudo encontramos que - hemos omitido algunas consideraciones básicas ó incluído algo extraño. Cuando esto ocurre, hemos repetido el proceso de análisis. En nuestro ejemplo --

./.

si decidimos incluír la pieza ENCONTRAR Y CORREGIR ERRORES (Parece improbable que nosotros no podamos), tendremos que analizar esa relación en sus piezas apropiadas. (ver figura 3.6.)

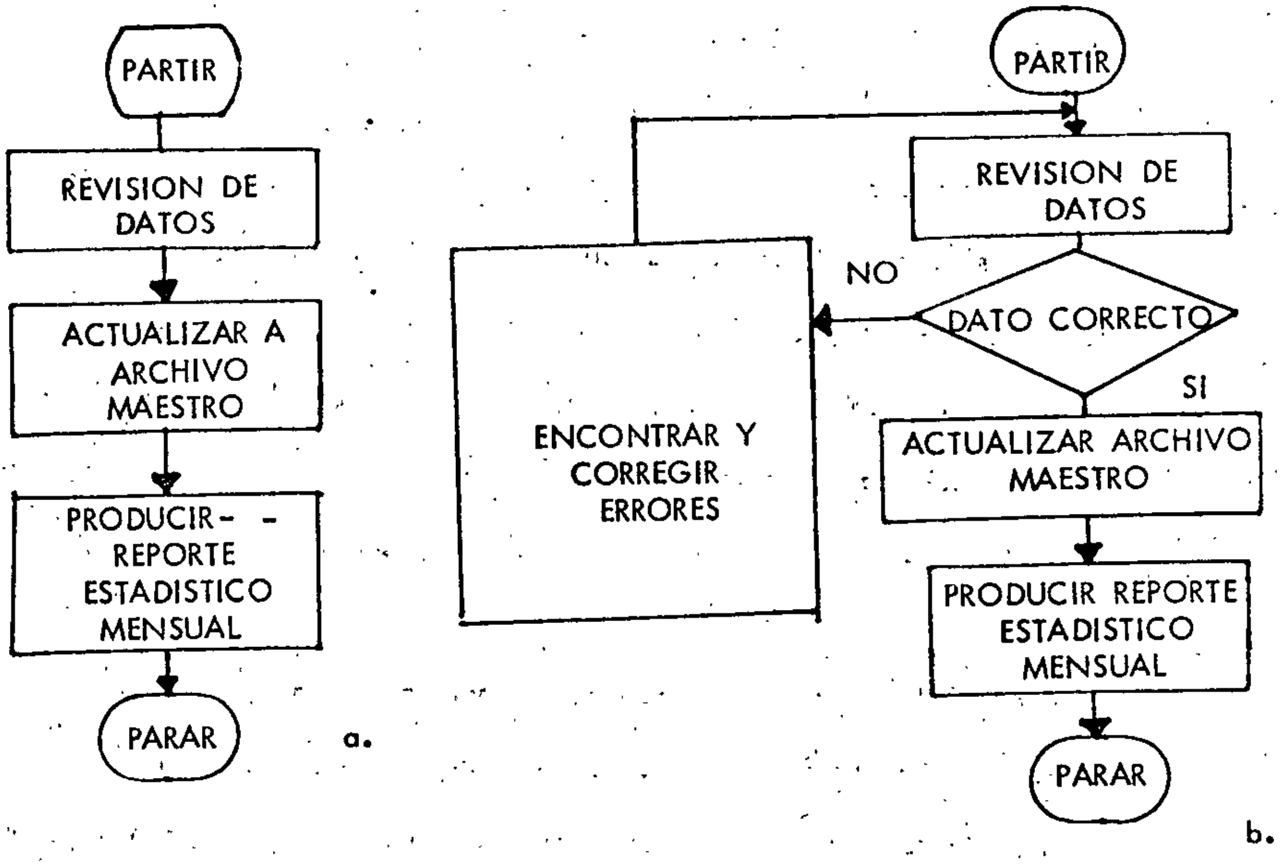
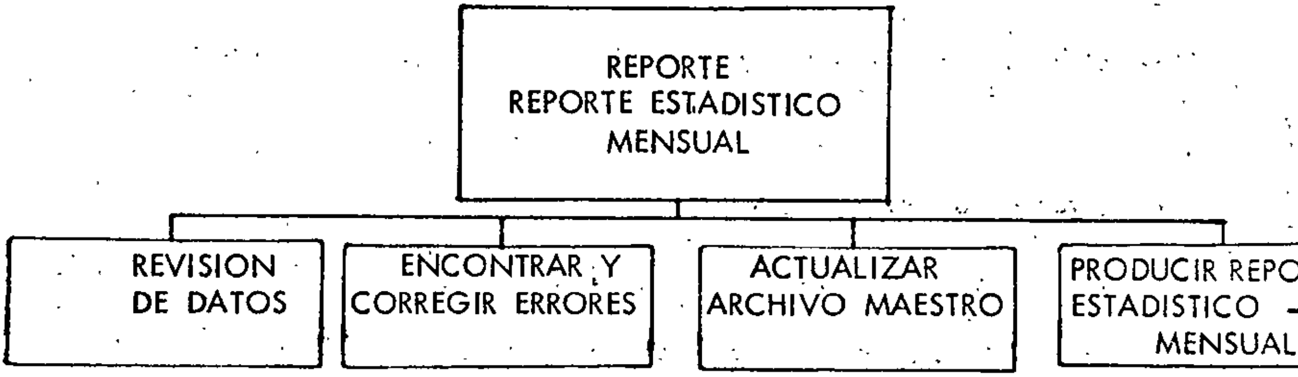

PARTIR
REVISION DE DATOS
ACTUALIZAR A ARCHIVO MAESTRO
PRODUCIR - REPORTE ESTADISTICO MENSUAL
PARAR
a.
PARTIR
REVISION DE DATOS
NO
DATO CORRECTO
SI
ENCONTRAR Y CORREGIR ERRORES
ACTUALIZAR ARCHIVO MAESTRO
PRODUCIR REPORTE ESTADISTICO MENSUAL
PARAR
b.

Figura 3.5. Comparando dos sistemas (a, b), con partes idénticas.

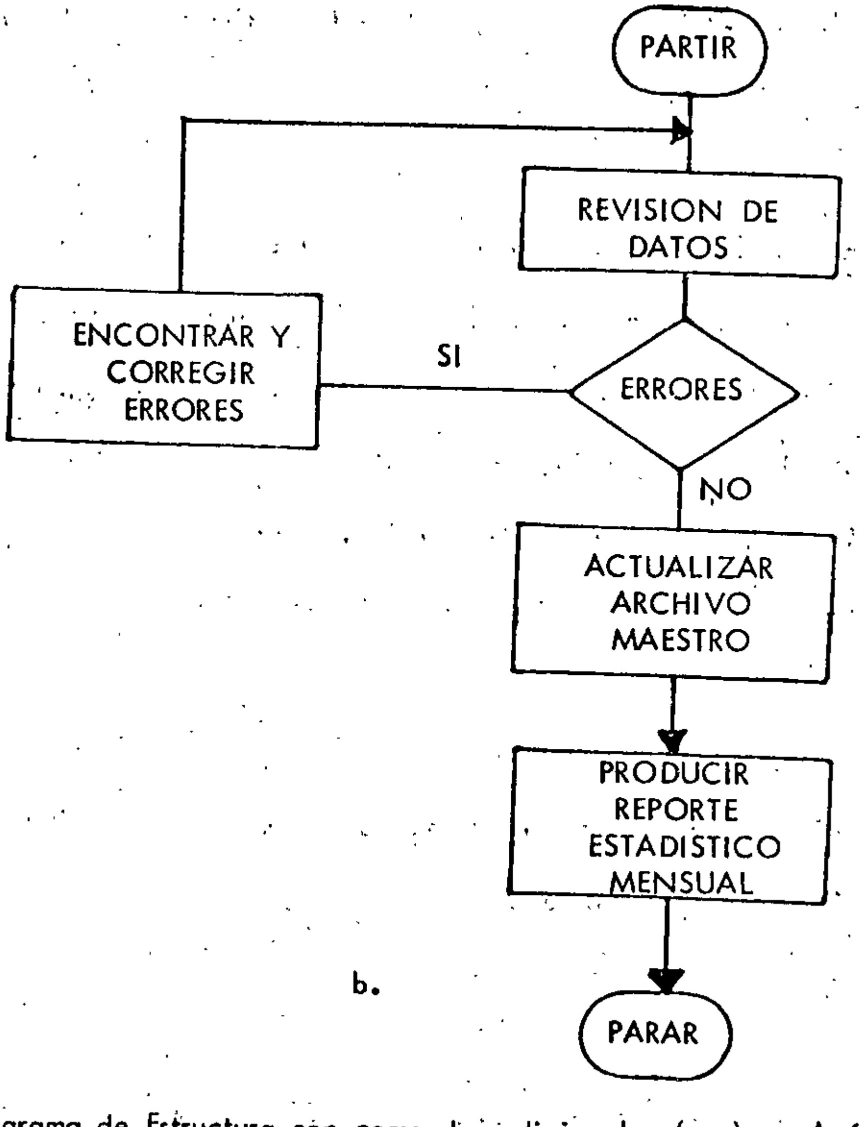
Anteriormente, mencionaremos brevemente la diferencia entre el diseño y el proceso de diseño. Ahora, nuestras palabras llegarán a ser más claras. Por una parte, después de colocar juntas las piezas otra vez se ha terminado el

./.

diseño, o al menos mejoraremos el que teníamos inicialmente.



a.



b.

FIGURA 3.6. Diagrama de Estructura con comandos adicionales (a) y, Análisis de aquéllo

A menos que el lector de tal diagrama tenga una copia de todos los pasos y - diagramas del análisis intermedio usados para desarrollar este diseño, el puede estar inclinado a pensar que llegamos con el diseño final la primera vez.

Muchas veces, los analistas que empiezan se frustran por el proceso de analizar, integrar, analizar, integrar, etc. Ellos sienten que deben estar haciendo algo erróneo, porque miran a otras personas terminar diseños y suponen que no tuvieron que ir por períodos de ensayo y error. Estos analistas principiantes - simplemente confunden el diseño con el proceso de diseñar. Hacer errores no es pecado; sin embargo, fallar al reconocerlos y tratar con ellos puede ser fatal.

Resientemente, hemos visto el proceso del análisis e integración (o síntesis), combinados en un enfoque simple llamado desarrollo de arriba-a-abajo. Como algo distinto del diseño de arriba-a-abajo, el desarrollo de arriba-a-abajo requiere no solo que su problema sea analizado en piezas, sino también que aquellas piezas sean construídas y probadas juntas como una estructura de la solu-- ción antes de que cualquiera de las piezas sean analizadas con gran detalle.

A fuerza de la integración continua, el desarrollo de arriba-a-abajo, se esfuerza por evitar muchos de los problemas más serios que han limitado el uso del - análisis jerárquico en el pasado.

./.

En lo que se lleva de esta sección, a propósito hemos tratado de evitar discusiones acerca de la estructura de los procesos de los sistemas o de usar el modelo explicado en el capítulo anterior. Quisimos concentrarnos en las maneras cómo diseñamos sistemas hoy en día, porque el diseño de sistemas es--- tructurados tiene mucho en común con los mfodos tradicionales.

En la mayor parte, la diferencia está en términos de énfasis y enfoque. Por lo tanto, antes de introducir cualquier estructura formal, quiero que el lector entienda los conceptos tradicionales de análisis y síntesis como es aplicado a la construcción de sistemas.

Tradicionalmente, el análisis y síntesis de sistemas se han usado junto con — otra técnica llamada " Flujo de Procesos ". Al mirar las cosas desde el punto de vista del flujo de procesos, usualmente intentamos averiguar cuando sucede qué.

Al analizar un problema, usamos los pasos del procesamiento corriente como - un indicio que nos dice que rompamos nuestro problema en piezas.

En la síntesis o integración, visualizamos que los pasos del proceso tendrán -- que tener que hacer el trabajo total.

./.

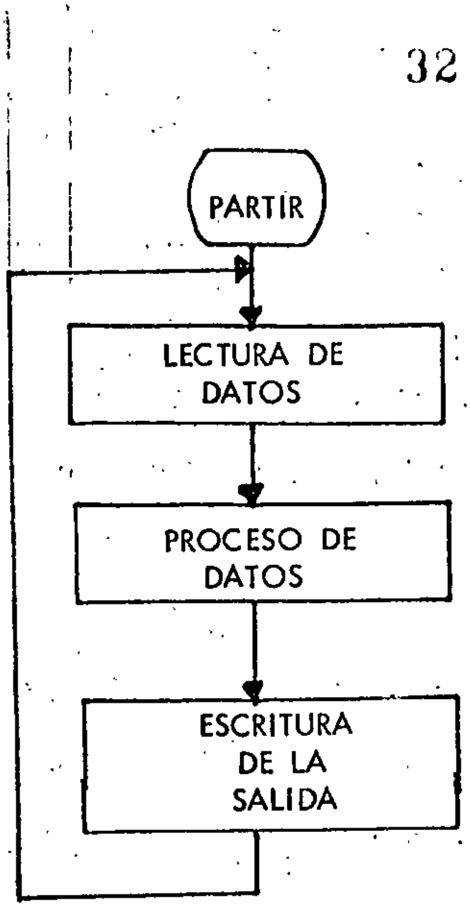


FIGURA 3.7. Modelo de Programación Básico.

Los programadores han usado el método de flujo de procesos como el medio primario para diseñar sus programas. En efecto, los diagramas de sistemas o diagramas de flujo de programas son todavía la herramienta de documentación de uso más común en cualquier parte.

La Figura 3.7. es tal vez el modelo de programación más común expresado en términos de un diagrama de flujo simple.

El flujo de procesos es un enfoque de diseño natural y forzoso, porque usted parte del principio y procede hasta el final.

Desafortunadamente, también tiene muchos defectos. Una cosa nos lleva a otra, pero procediendo de un diagrama de flujo de procesos general a uno - más detallado, puede llevarnos a diseñar problemas. Por ejemplo, en el programa simple dado en la hoja anterior, existe una iteración infinita, o error, dado que el programa falla al procurar un medio de parar, i. e., simplemente parece ignorar la señal de salida de los datos. Para hacer esté modelo -- trabajable, se requiere entonces la siguiente significación :

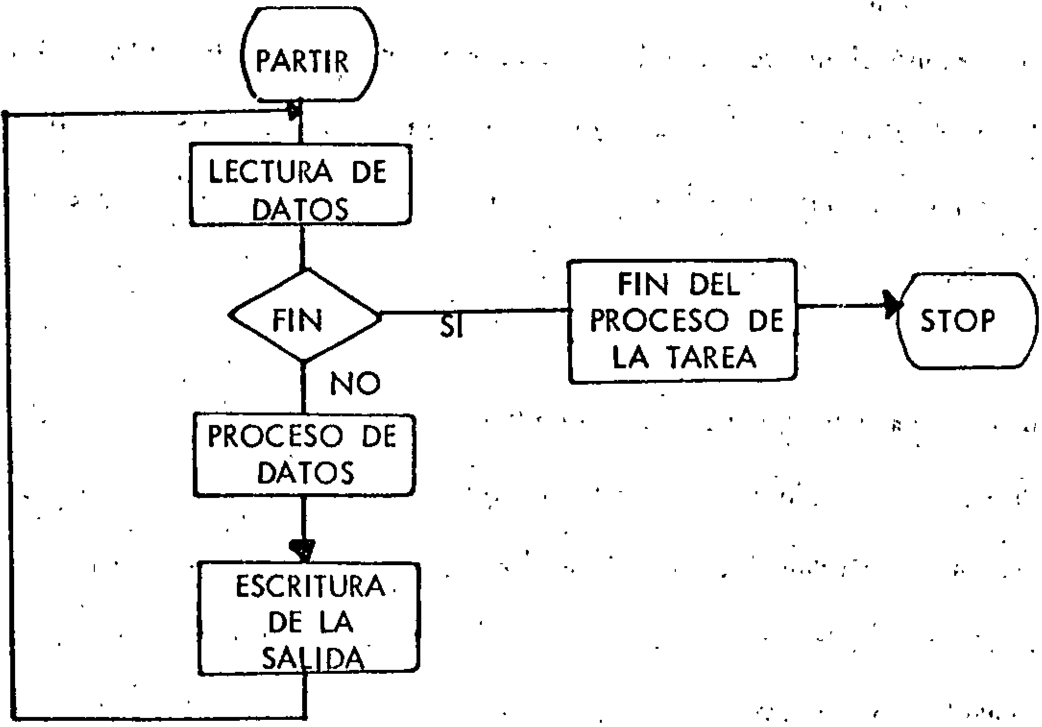


FIGURA 3.8. Modelo de programación Básico Corregido.

Es una característica inevitable del flujo de procesos, diseñar aquélla, con nuevas condiciones lógicas que se manifiestan inesperadamente (llamándolas - excepciones), el flujo de procesos básico total llega a ser más difícil de reconocer.

Así como aprendamos más acerca del desarrollo de programas, encontramos menos atractivo el flujo de procesos como un método de diseñar, ya sea un programa o un sistema.

Debido a que el diseño de flujo de procesos llega a ser extremadamente complejo, y dado que diseñar con flujo de procesos es fundamentalmente un método de ensayo y error, usualmente terminaremos con mucho más error del que es permitido.

Pero no importa lo que nuestros sentimientos sienten acerca del diseño de flujo de sistemas, cualquier sistema terminado debe asegurar que los procesos - convenientes sucedieron en los tiempos correctos dentro de un sistema. Afortunadamente, la estructura jerárquica hace esto posible por caminos nunca - sospechados unos cuantos años antes. Al mismo tiempo, elimina mucha de - la confusión y complejidad del flujo de procesos del diseño de sistemas de - los primeros tiempos.

./.

Hemos visto unas cuantas de las herramientas principales del diseño de sistemas tradicional: análisis, síntesis (o integración) y, flujo de procesos de sistemas.

Colocamos estos elementos juntos en una nueva forma a la cual se le refirió al principio como diseño de sistemas estructurados. Por otra parte, este método nos permite combinar las herramientas del análisis, integración y flujo de procesos de una manera lógica y aprovecharse de nuestro modelo básico de un sistema.

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

ARQUITECTURA DEL SISTEMA
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

1

VII. ARQUITECTURA DEL SISTEMA (DOCUMENTACION COMPLEMENTARIA)

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

Comparing Software Design Methodologies

2

by Lawrence J. Peters and Leonard L. Tripp

Software design has evolved to the stage where methodologies for handling classes of problems are proliferating. But just how helpful are they?

In progressing from infancy to adulthood, a human's approach to problem solving shifts dramatically. In infancy, a challenge such as locomotion is treated as new and different each time it is faced. After a while, it is recognized that a certain class of challenges can be met using the same approach. Similarly in software we have grown from treating each new development effort as unique to recognizing certain classes can be met by a specific approach. This type of evolutionary process quite evidently has already occurred in other professions such as architecture and engineering.

The last ten years in the software industry have been marked by a procession of new approaches to software design problems. The cause of this influx of software design methods is uncertain. Perhaps it is part of evolution or it may be due to the increasing complexity of the problems being addressed. In any event, the availability of so many approaches has left many wondering which—if any—they should adopt, which ones fit which classes of problems.

We've asked ourselves that question. We've studied several of the more promising or more popular approaches. And we think we can provide at least a partial answer.

Where do I begin? Now that I have begun, how do I measure my progress? How will I know when I am done? These questions have always tormented designers. Designers are also vexed by having to think intuitively, rationally, and procedurally at the same time during a design effort. As the effort progresses, the emphasis shifts, but all three modes often are involved simultaneously. At the outset, the designer initiates some idea or "spark" which sets a design into motion. He has an intuitive feel for the solution to the problem but suspects he may be wrong. So he scrutinizes his idea and then documents the conclusions. This process has been characterized as divergence, transformation, and convergence, in that order. The big problem is broken down into smaller problems, they are solved, and then reassembled into *the* solution. Simple isn't it?

Not so, say those who have attempted to develop non-trivial designs. Some even conjecture that software designing is somewhat diabolical. On examining the process of design, it becomes apparent that there is no agreement on how to describe that process and/or its products. The following definition may be a helpful start; at least it works for several individual methods in the latest procession: *A software design method is a collection of techniques based upon a concept.*

Many forms of this notion are being championed. A representative list includes:

- Structured Design
- The Jackson Methodology
- Logical Construction of Programs
- METAStepwise Refinement (MSR)
- Higher Order Software (HOS)

The author of each such software design method has structured his solution to address the design issue(s) he views as germane. Quite understandably, each holds a different opinion. Those that advocate "Structured Design" declare that the key to a successful software design is the identification of the data flow through the system and the transformation(s) that the input data undergo in the process of becoming output.

A view held by those who advocate either the "Jackson Methodology" or the "Logical Construction of Programs" (the "Warnier Methodology") is that the identification of the inherent data structure is vital, and the structure of the data (input and output) can be used to derive the structure (and some details) of the program.

Advocates of META Stepwise Refinement (MSR) state that success is assured if the problem is solved several times, each solution being more detailed and complete than its predecessor.

Last, supporters for Higher Order Software (HOS) provide a set of axioms which must be used to attain success.

In addition to making certain assumptions, each of the representative methods also prescribes a set of activities and techniques intended to ensure successful software design.

Structured design

Structured Design is based on concepts originated by Larry L. Constantine and later published by him while working with Ed Yourdon, and by Glenford J. Myers. (See Bibliography for exact references.)

The method consists of concepts, measures, analysis techniques, guidelines, rules-of-thumb, notation, and terminology. Reliance is placed upon following the flow of data through the system to formulate program design. The data flow is depicted through a special notational scheme which identifies each data transformation, transforming process, and the order of their occurrence.

The interpretation of the system specification is used to produce the data flow diagram, the diagram used to develop the structure chart, the structure chart to develop the data structure, and all of the results used to reinterpret the system specification. While the design process is iterative, the order of iteration is not rigid.

The process seems deceptively simple; but when attempts are made to use it, difficulties are encountered. For example, consistently identifying transformations of data is not easy to do. It is possible to be overly detailed in one part of the data flow and much less so in another. No formula is available to detect this condition.

Also, identifying afferent (incoming) and efferent (outgoing) flow boundaries plays an important role in the definition of the modules and their relationships. However, the boundaries of the modules can be moved *almost* arbitrarily, leading to different system structures. Again, no formal guide is provided.

Use of the structured design method does aid in the rapid definition and refinement of the data flows. The verification of the consistency of one data flow with its less detailed predecessor is crucial to this, but how to do the verif cation is not satisfactorily addressed. Admittedly, this activity has been addressed for a military command and control application we learned of, but the technique used there is not an integral part of the structured design method. We find that this method and particularly its graphics do reveal previously unknown properties of some systems, though, such as the generation of information already contained elsewhere in the system.

This method turns out to be well suited to design problems where a well-defined data flow can be derived from the problem specifications. We found that some of the characteristics that make a data flow "well-defined" are that input and output are clearly distinguished from each other, and that transformations of data are done in incremental steps—that is, single transformations do not produce major changes in the character of the data.

Jackson methodology

The Jackson Methodology views data structure as a driving force to successful software design. The method was popularized in England through the efforts of Michael Jackson, hence its name, and more recently through efforts of Infotech Information Ltd.

In this methodology a program is viewed as the means by which input data are transformed into output data. An explicit assumption is that paralleling the structure of the input (data) and output (report) will ensure a quality design. One *implicit* assumption is that the resulting data structure will be compatible with rational program structure. Other implicit assumptions include that only serial files will be involved and that the user of the method knows how to structure data.

Some claimed characteristics of this method include:

- It is not dependent on an analyst's experience or creativity.
- It is based on principles by which each design step can be verified.
- It is not difficult to learn and use correctly. (So if given the same problem, two designers working independently would arrive at very nearly the same design.)
- It results in designs which are easy and practical to implement.

Again, the process seems simple. But when we attempted to employ this technique, several difficulties were encountered. One was with the supporting documentation (Jackson's book) which is laden with examples and too few explanatory notes. We also encountered problems with the practicality of the method and began to question its basic premises. To illustrate, error processing had to be "wedged in" as erroneous data do not exist in a structural sense. Also, various file accessing and manipulation schemes took their toll. For instance, much data structuring is dictated by the data base management system employed. Thus, whether the data are tree-structured or not, we still may end up with an unimplementable program because there

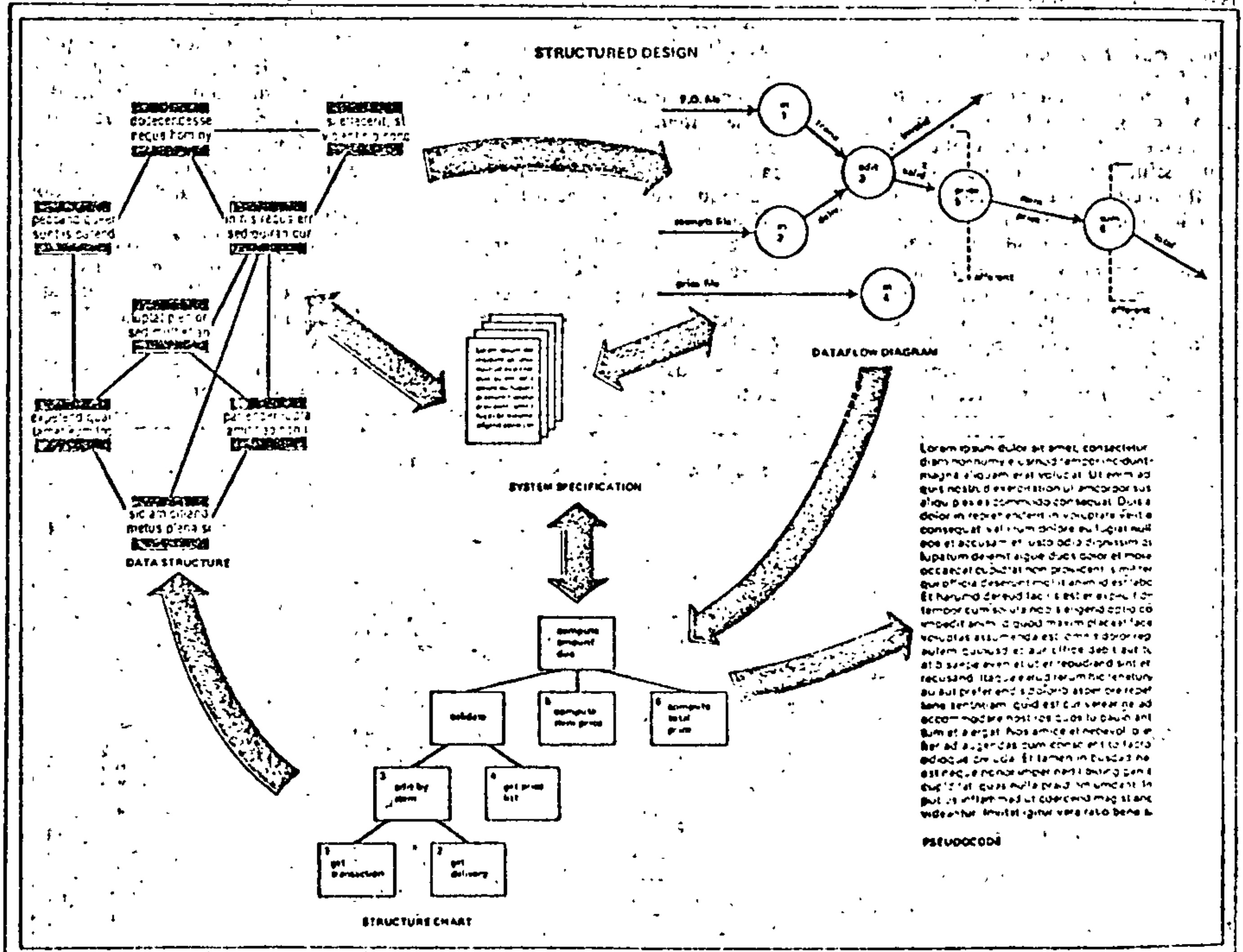


Fig. 1. Advocates of Structured Design hold that the flow of data through a system is the key to program design. The system specification is used to produce the data flow diagram, the diagram to develop the data structure chart, the chart to develop the data structure, and all of the pieces to reinterpret the system specification.

As might be expected, the methodology works best where input data are transformed into output in incremental, easy to follow steps.

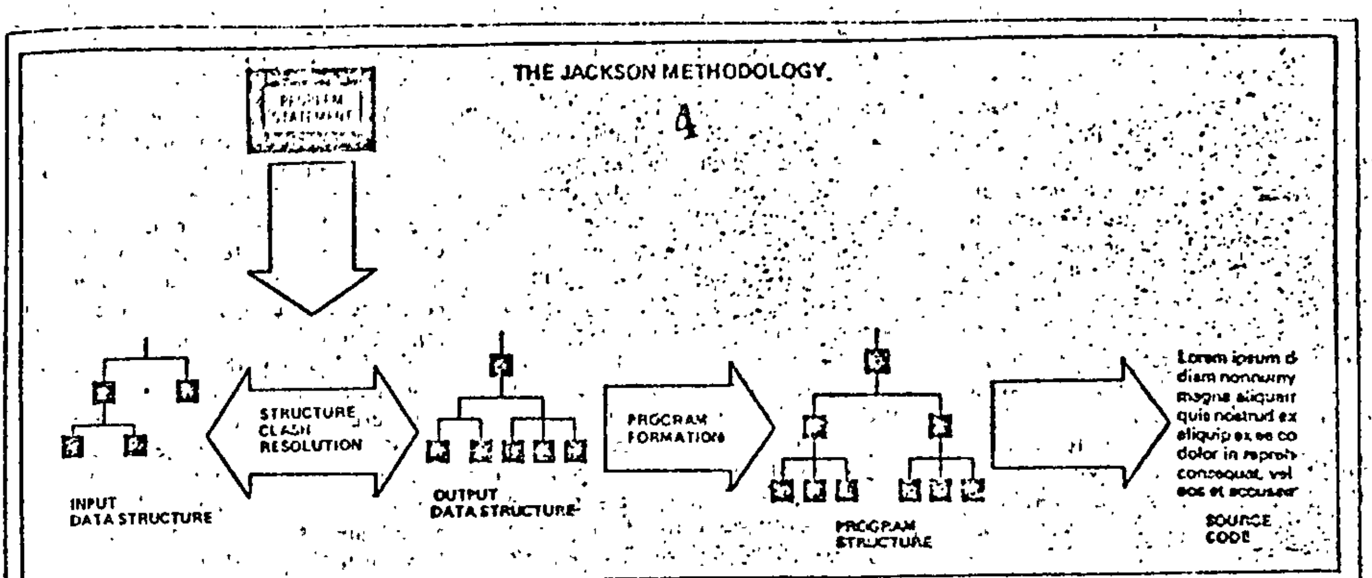


Fig. 2. The Jackson Methodology assumes that making the output data structure parallel to the structure of the input data will lead to a "good" design. (Some of the other methodologies share this assumption, too.) Unfortunately, the method may be practically limited to serial files. Also unfortunately, there may be no causal link between data structure and program quality.

causal

does not appear to be a casual link between data structure and program quality; the basic assumption is invalid.

Logical construction of programs

LCP is a method similar in nature to Jackson's design methodology in that it also assumes data structure is the key to successful software design. However, this method is more proceduralized in its approach to program design than the Jackson method. Originated by Jean-Dominique Warnier in France, it has become popular outside the United States. It will now have its opportunity to become popular with English-speaking designers as English translations of the original French text are now available and the method has been incorporated into training offered by Infotech.

The LCP method is as follows:

1. Identify all input data to the software and organize in a hierarchical manner (files, records, entries, items). The exact format is not of concern here, but rather how the various parts of the input file are related to one another.
2. Define and note the number of times each element of the input file occurs, using variable names to relate the ratio of occurrences (such as: one active customer file, N customer records, each customer record has four entries: address, most recent payment, current balance, new charges).
3. Do step 1 and step 2 above for the desired output.
4. Obtain the details of the program by identifying the types of instructions contained in the design in a specific order: read instructions, preparation and execution of branches, calculations, outputs, and subroutine calls.
5. In flowchart-like fashion, depict the logical sequence of instructions using "Begin Process," "End Process," "Branch," and "Nesting" indicators.
6. Number the elements of the logical sequence and expand each

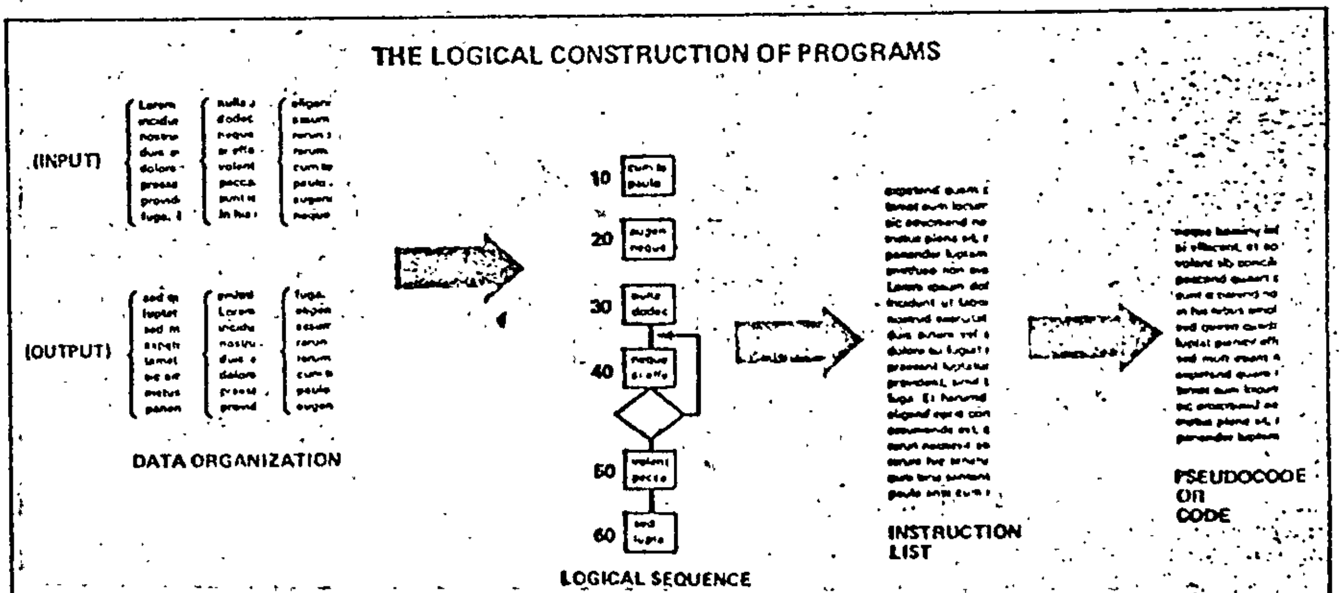


Fig. 3. The "Logical Construction of Programs" has become more popular outside of the U.S. than within its borders, possibly due to the lack of good English language documentation. The method again involves structuring the input and output data, using its own graphic conventions (Warnier diagrams), but adds a set of sequencing procedures and instruction types to translate the logical arrangement into pseudocode.

through the instructions identified in Step 4. (There exist several other guidelines regarding how data structure conflicts are involved, but they are not pertinent to our discussion.)

Many of the difficulties associated with the use of this method are similar to those encountered in using the Jackson Methodology. For instance, some problems force us to contrive a hierarchical data structure where none was previously apparent. Also, this method is somewhat misnamed in that it deals with program design issues and not construction issues (such as packaging, run environment, file access methods, etc.). Although for a problem with a readily apparent hierarchical data structure we get to a pseudocode statement of the program very rapidly, closer inspection often reveals that the resulting program is not what we would have chosen.

This method appears to be well suited to problems involving one module or only a few modules, and where the data are tree-structured. The latter leaves it susceptible to the same kind of problems as the Jackson Methodology.

Meta stepwise refinement

MSR is based on the premise that the more times you do something, the better the final results. It allows the designer to assume a simple solution to a problem and gradually build in more and more detail until the complete, detailed solution is derived. Several refinements, all at the same level of detail, are conjured up by the designer each time additional detail is desired. The "best" of these is selected, more detailed versions proposed, the best of these selected, and so on. Only the *best* solution is refined at each level of detail. Specific attributes of this method include:

1. It requires an exact, fixed problem definition.
2. It is programming language independent in early stages.
3. Design is by levels.
4. Details are postponed to lower levels.
5. Correctness is ensured at each level.
6. The design is successively refined.

MSR was authored by Henry Ledgard and later given this name by Ben Schneiderman. It is a synergism of Mill's top-down notions, Wirth's stepwise refinement, and Dijkstra's level structuring. It produces a level-structured, tree-structured program.

It is well known that by proper program organization it is possible to separate functionally independent levels

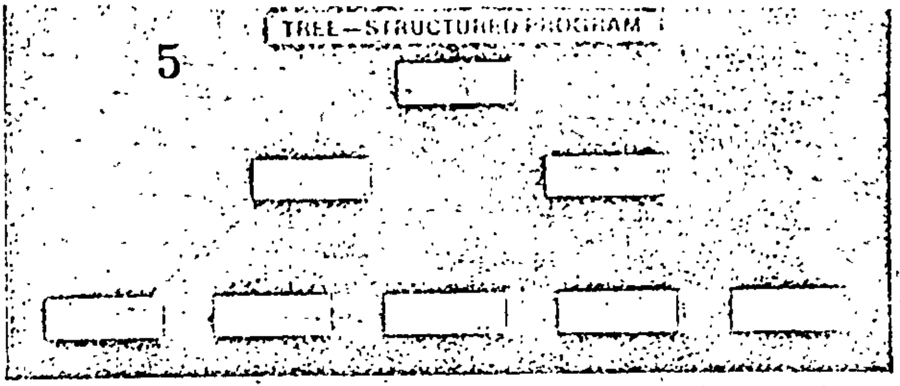


Fig. 4a. In a tree-structured program, the "root" module contains an outline or general image of the program, while the lower levels contain increased amounts of implementation detail.

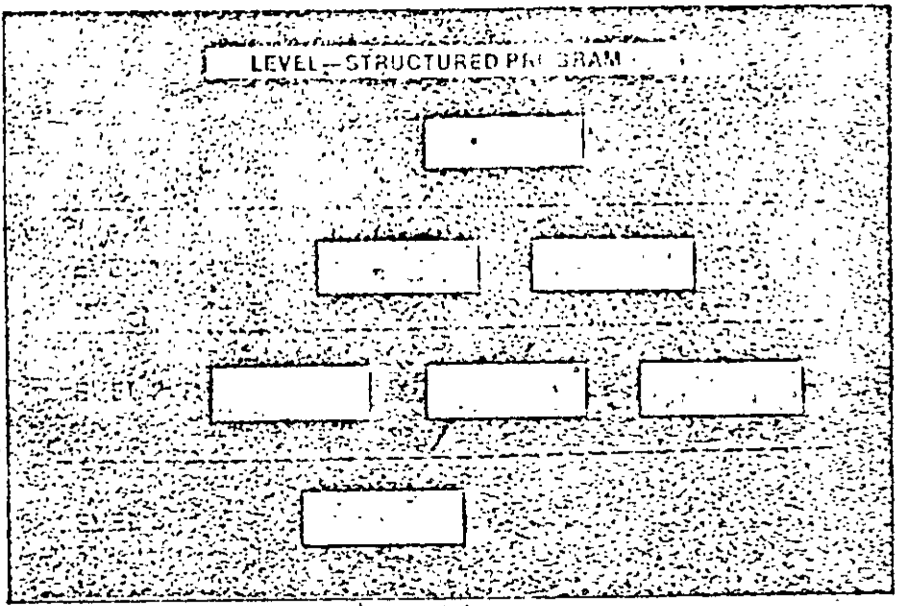


Fig. 4b. The basic rule of organization in a level-structured program is that modules at one specific level invoke only modules at the next lower level and never the reverse.

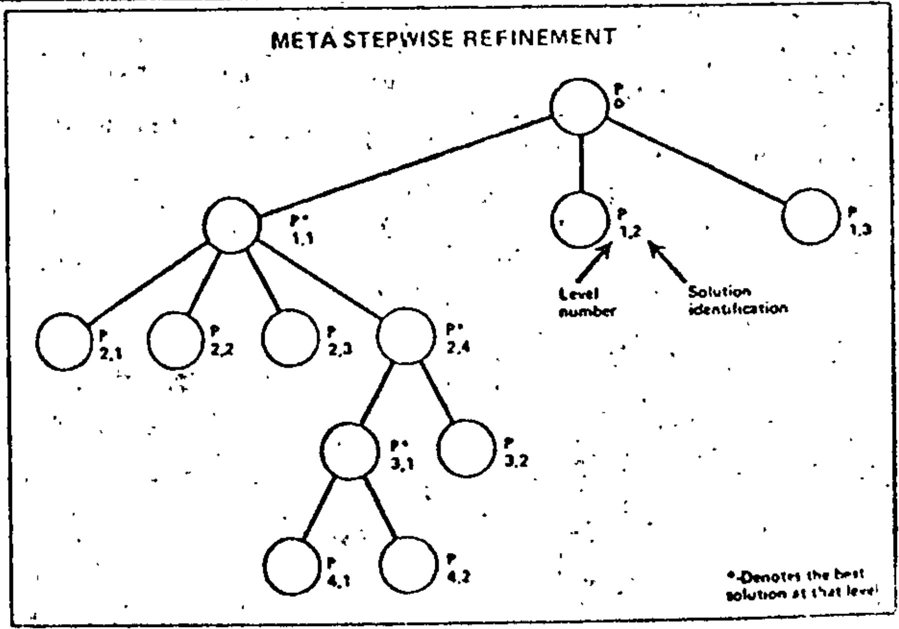


Fig. 5. In using Meta Stepwise Refinement, the designer starts with a simple, general solution and builds in increasing amounts of detail at lower levels in the design. MSR requires that the designer actually develop several potential solutions at each level, discarding all but the best of these. It may take a special kind of person to do this.

6

METHOD	Requirements Analysis and Specification			Design			Implementation	
	Problem Definition	Value System Design	Systems Analysis	Architecture Design	Data Design	Module Design	Code Construction	Testing
STRUCTURED DESIGN				1	2	2		
THE JACKSON METHODOLOGY				1-2	2	1	1	
LOGICAL CONSTRUCTION OF PROGRAMS					2	1	1	
META-STEPWISE REFINEMENT				2		1	1	
HIGHER ORDER SOFTWARE	2			3		1	1	

1=addressed directly   2=covered but no substantive guidelines offered   3=supported by automated processor(s)   blank=does not cover

or layers in programs (as is illustrated in Figs. 4a and 4b). The higher levels reflect the problem statement while the lower ones have increasing amounts of implementation detail. The basic rule is that modules at a specific level invoke only modules at the next lower level, and never the reverse. This is how MSR works.

Although the method's theory sounds good, the practice leaves a lot to be desired. For example, in real life, non-trivial problems undergo constant reinterpretation, reevaluation, and modification. They are not stable. If they were, this method would not be needed. Since the solution at any one level depends on prior (higher) levels, and since any change in the problem statement affects prior levels, our ability to produce a solution at any level is undermined until the changes are made.

One approach is to refuse changes until the design is complete. This results in the solution and the requirements being unsynchronized. The production of multiple solutions is another difficulty. Coming up with fundamentally different solutions to a problem is not a likely occurrence for an individual. Also, how to decide which solution is "best" is not addressed by this method.

Due to the number of times the problem is going to be solved, this approach works best on small problems, perhaps those involving only a single module. It is particularly useful where the problem specifications are fixed and an elegant solution is required, as in developing an executive for an operating system.

Higher order software

HOS initially was developed and promoted by Margaret Hamilton and Saydean Zeldin while working on NASA projects at MIT. The method was invented in response to the need for a formal means of defining reliable, large scale, multiprogrammed, multiprocessor systems. Its basic elements include:

1. a set of formal laws
2. a specification language
3. an automated analysis of the system interfaces
4. layers of system architecture produced from the analyzer output
5. transparent hardware

This design method is based on axioms which explicitly define a hierarchy of software control, wherein control is a formally specified effect of one software object on another:

Axiom 1: A given module controls the invocation of the set of valid functions on its immediate, and only its immediate, lower level.

Axiom 2: A given module is responsible for elements of only its own output space.

Axiom 3: A given module controls the access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate, lower level function.

Axiom 4: A given module controls the access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

Axiom 5: A given module can reject invalid elements of its own, and only its own, input set.

Axiom 6: A given module controls the ordering of each tree for the immediate, and only the immediate, lower levels.

In practice, HOS has been used with an automated analyzer program which checks the solution design as expressed in HOS' own metalanguage. The need for the analyzer is not inherent in the methodology, however, and in fact we used pseudocode in examining the method since the analyzer was not available to us.

Our evaluation is that HOS is an asset in applications where the accuracy and particularly auditability of the algorithm are the primary concerns, applications such as scientific problems and detailed financial computations.

Our exposure to this methodology is limited, due in part to its very nature, its scale. We did use it enough to find some of its characteristics, however. For example, one objective of HOS seems to be to ensure correctness and consistency by interface definition and attention to detail. It was this emphasis on the details of system execution which first focused our attention on a

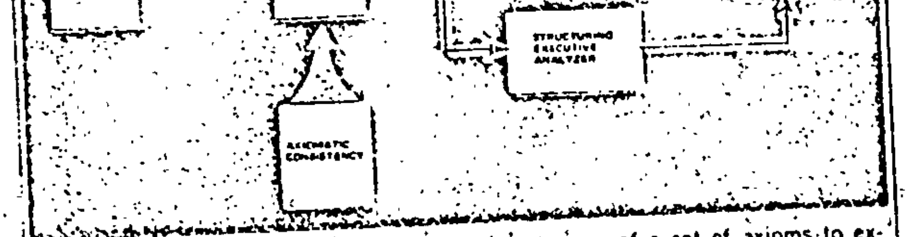


Fig. 6. Higher Order Software Design involves the use of a set of axioms to explicitly define the hierarchy of software control. It has been used with analyzer programs which check the design solution as expressed in HOS' own metalanguage. Developed for large scale NASA projects, the methodology may be well suited for operating system design, but it does not address the structure of

7

METHOD	ATTRIBUTE							
	Specialized Graphics	Defined Procedure(s)	Training Support	Tutorial Documents	Requirements Traceability	Known Experience Base	Compatibility With Other Techniques/ Schemes	Area of Application	Evaluation (Quality) Criteria
STRUCTURED DESIGN	use structure charts for system architecture	an iterative framework which guides the solution development	two courses offered by Yourdon, Inc.	book by Yourdon & Constantine, book by G. J. Myers	designer's responsibility	up to 5 years experience within firms like IBM & Hughes	usable with any module design strategy	systems whose data flow can be communicated graphically	a well-defined set of design heuristics
THE JACKSON METHODOLOGY	tree-like charts for data structures	loosely defined guidelines to address various problems	two-week course offered through Infotech	book by Jackson (challenge to read) presented via examples	designer's responsibility	early versions available since 1972 with emphasis on business application	usable with other data structuring methods	business & other systems with well-understood data structure(s)	verify compliance with basic assumptions
LOGICAL CONSTPUCTION OF PROGRAMS	use Warnier chart for data structure	well-defined set of procedures at all levels of detail	incorporated in a course offered by Infotech	book by J.-D. Warnier	designer's responsibility	extensive use throughout Europe & other foreign countries	procedural nature would limit compatibility	business & other systems with well-understood data structure(s)	verify compliance with basic assumptions
META-STEPWISE REFINEMENT	use a tree diagram for program structure	high-level guidelines for the basic steps	no formal offerings	book by H. F. Ledgard	designer's responsibility	primarily limited to theoretical developments	would benefit from design evaluation criteria	applications with well-understood, stable requirements	no specific guidelines
HIGHER ORDER SOFTWARE	structured flowcharts for control structure	mostly theoretical discussion(s) with limited operational details	by arrangement with Higher Order Software	no formal text, several papers in journals	potentially available through an analyzer	proposed application on NASA Space Shuttle program	would benefit from design guidelines	applications with high reliability requirements	primarily automated analysis of design

fraction of the software design problem: we found that the issue of data base design was, at best, addressed implicitly, with the structure of the code appearing to Le the primary problem Our experience with large systems has taught us that the design of codt and data base must be synchronous.

Now what do we know?

Now that we've gone through all the experimentation, what have we learned?

The preceding remarks are based on our experiences and reflect personal biases, but four observations should be apparent by now. The first is that no single method exists which would be an asset in every design problem. That's no surprise. The second observation is that the assumptions made by each method are just that—things taken for granted and not provable. The third is that methods can only contribute so much to the design effort. Designers produce designs, methods do not. A design problem, although well suited to a particular technique, will always have some quirk which makes it unique. Software design methods merely assist in solving routine aspects of a problem. Using a methodology only reveals the critical issues in a design effort and gives us more time to address them.

The final observation is that designing is problem solving—a fundamental, personal issue. To many, design methods are something of an affront and are resisted if imposed. Adoptioo of a method or methods requires a behavioral change, an alteration in how problems (of a certain class) are solved. And accomplishing the desired behavioral change can be a very difficult undertaking.

Methods are important but their successful application occurs only in supportive environments. Specifically, the necessary management elements (planning, scheduling, control systems, etc.) must all be present and effective. The larger the system, the more important these "non-technical" factors. The balance between methods and environments is a delicate one. The merging of these may very well be the next evolutionary step. ✿

Bibliography

Bazjanac, V. "Architectural Design Theory: Models of the Design Process," in *Basic Questions of Design Theory*, W.R. Spillers, editor. (New York: American Elsevier Publishing Co., Inc., 1974).

Hamilton, M., and Zeldin, S. "Higher Order Software—A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 9-31.

Jackson, M.A. *Principles of Program Design* (New York: Academic Press, 1975).

Jensen, E.P. *1975 IR & D Structural Design Methodology—Volume II* (Hughes Aircraft Co., December, 1975).

Jones, J.C. *Design Methods* (New York: Wiley-Interscience, 1970).

Ledgard, H.F. "The Case for Structured Programming, *Bit*, Vol. 13, 1973, pp. 45-57.

Myers, Glenford J. *Reliable Software Through Composite Design* (New York: Petrocelli Charter, 1975).

Peters, L.J., and Tripp, L.L. "Is Software Design Wicked?," *Datamation*, Vol. 22, No. 6, June 1976.

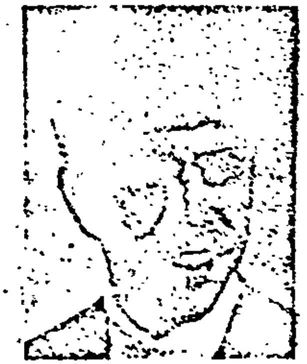
Rittel, H.W.J., and Webber, M.M. "Dilemmas in a General Theory of Planning, Working Paper No. 194," (Berkeley: Institute of Regional Development, University of California, Nov., 1972).

Schneiderman, Ben. "A Review of Design Techniques for Programs and Data," *Software—Practice and Experience*, Vol. 6, pp. 555-567, Chichester, England, 1976.

Stevens, W.P., Myers, G.J., and Constantine, L.L. *Structural Design* (New York: Yourdon, Inc, 1975).

Warnier, J.-D. *Logical Construction of Programs* (Leiden: H.E. Stenfert Kroese B.V., 1974).

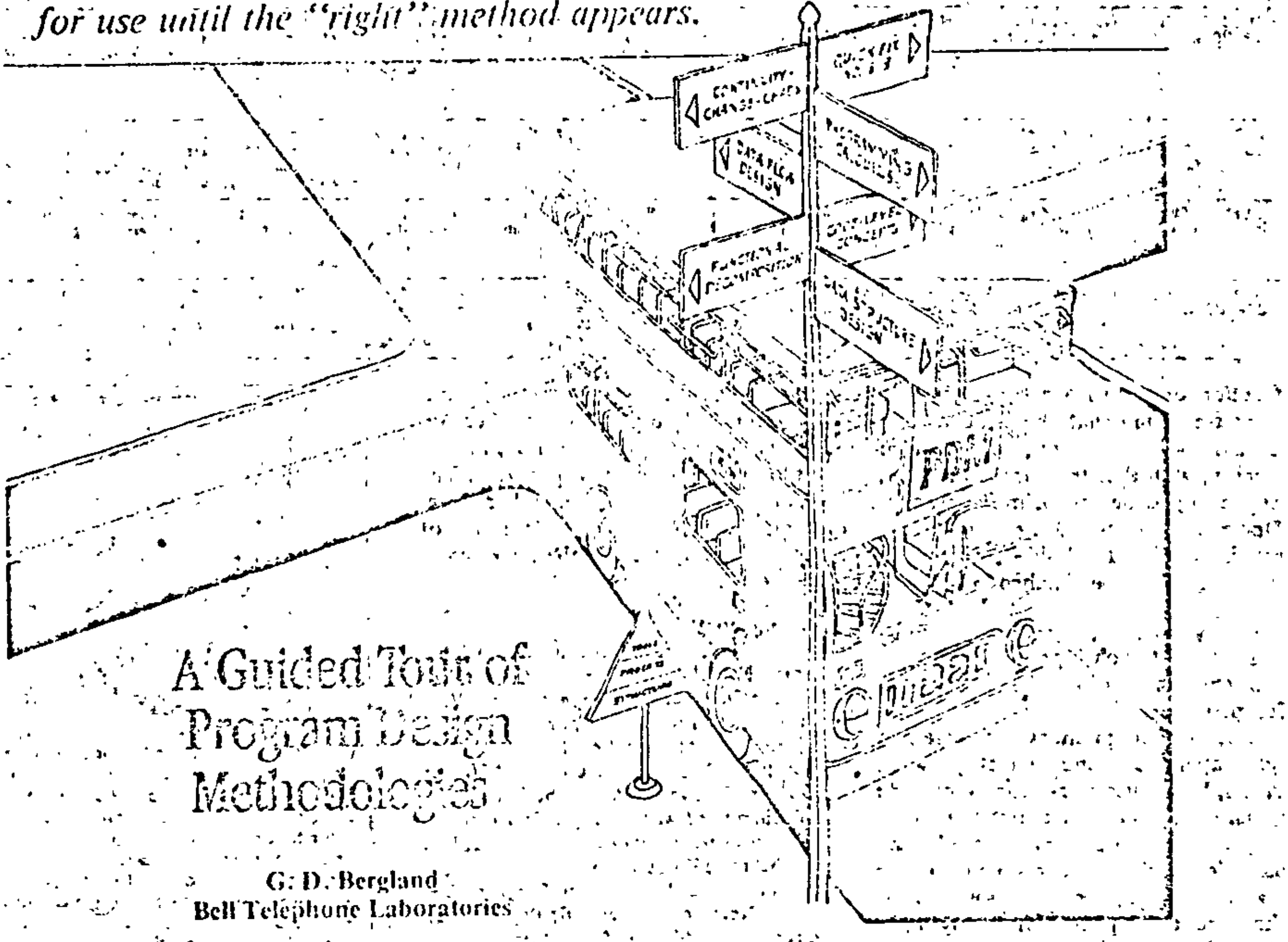
Yourdon, E., and Constantine, L.L. *Structured Design* (New York: Yourdon, Inc., 1975).



Mr. Peters is a software engineering consultant with Boeing Computer Services Inc. His ten years of dp experience include periods spent on commercial, scientific, and real-time applications while working at Aerojet Electrosystems, Bell-Northern Research (in Canada), and BCS. Since 1973 he has been involved in the development and promotion of modern programming practices within BCS.

Mr. Tripp started as a software engineer (although the title wasn't yet invented) at the Boeing Company in 1966, working on large matrix structural analysis systems and on non-linear differential solving. He too has been working on the programming practices project at BCS since 1973, lately in his capacity as manager of software methods development and implementation within the Advanced Techniques and Applications Div.

After describing, applying, comparing, and evaluating four major methodologies, this guided tour concludes with an interim procedure for use until the "right" method appears.



A Guided Tour of Program Design Methodologies

G. D. Bergland
Bell Telephone Laboratories

Much as a building architect specifies the structure and construction of a building (see Figure 1), the software architect must specify the structure and construction of a program. This guided tour examines some of the concepts, techniques, and methodologies that can aid in this task.

During this guided tour, the software problem and the attempts at its solution are briefly described. Software engineering techniques are classified into three groups: those that primarily impact the program structure, the development process, and the development support tools. Structural analysis concepts are described that have their major impact at the code level, the module level, and the system level. Then, four of the major program design methodologies that have been reported in the literature are developed and compared. Functional decomposition, data flow design, data structure design, and programming calculus are described, characterized, and applied to a specific example.

While no one design methodology can be shown to be "correct" for all types of problems, these four methodologies can cover a variety of applications. Finally, an interim approach for large software design problems is suggested that may be useful until an accepted "correct" methodology comes along.

Motivation. The major motivation for looking at program design methodologies is the desire to reduce the cost of producing and maintaining software. Developing programs that are reliable enough to support nonstop computer systems can sometimes be a secondary motivation, but these applications seem to be in the minority.

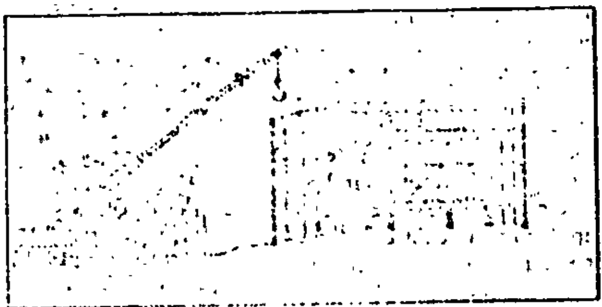


Figure 1. A good design implies a good structure.

A host of claims in technical journals, conference proceedings, and short-course advertisements herald the virtues of new design methodologies—claims such as 1.9 to 1 increase in productivity, 39 percent savings in programming costs, 80 percent reduction in bugs, etc. Their inconsistencies suggest that without a good set of metrics, you can prove anything you want. Alternatively, software development is so inefficient that almost anything can improve it.

If one were to take these claims at face value, it would seem that at least some of the problems of producing inexpensive, reliable software have been solved. Unfortunately, the benefits of structured programming, software engineering techniques, or whatever have remained either nebulous or illusive to many people. Even though structured programming has been with us for more than a decade, we are still far from having all the answers or, for that matter, even all of the questions. While it is clear that progress has been made, there is still much to be done.

Historical perspective. During the 1950's, programming was in its golden age. The approach was to take a small group of highly qualified people and solve a problem by writing largely undocumented code maintained by the people who wrote it. The result was inflexible and inextensible code, but it was adequate to the demands of the time. Buxton[1] called this "cottage industry" programming.

The software crisis hit in the 1960's. The problems got two orders of magnitude harder, and we were introduced to the problems of having many people work on large programs that were continually changing. This was the beginning of "heavy industry" programming.[1]

The structured programming of the 1970's was primarily an attempt to address the problems of heavy industry programming. This quest for a better way was started in response to rapidly rising costs[2]—more than one percent of the gross national product was being spent on software in the US—and the feeling that change was technically feasible. Thus started a variety of approaches that are collectively known as structured programming.

Techniques hierarchy. Software engineering has been defined by Parnas as multiperson construction of multiversion programs.[3] As such, there is much emphasis on the development process, its attendant support tools, and the basic structure of a program. Many of the concepts which people tend to apply first—like teams, design reviews, and program librarians—primarily involve changing the development process. Vyssotsky characterized these techniques as dealing with programmer "crowd control."[4] While they can be implemented relatively quickly, their major benefits can only be realized in the context of a well-structured program. Those techniques dealing with program structure form the foundation on which the other techniques should be applied[5] (see Figure 2). Admittedly, the support tools and the development process strongly influence the structure of the program; however, the tools should be adapted to support the desired structure, not vice versa.

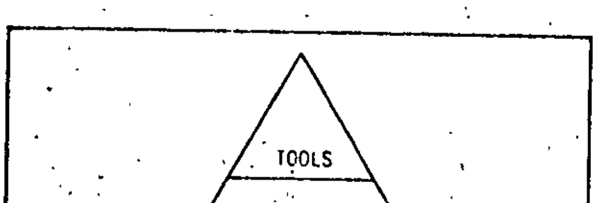


Figure 2: Software engineering techniques hierarchy.

While there are many design methodologies around, only a few of them have been extensively tested. Four methodologies used or discussed more than most are

- functional decomposition,
- data flow design,
- data structure design, and
- programming calculus.

I believe that the quality of the program structure resulting from a design methodology is the single most important determinant of the life-cycle costs for the resulting software system. Thus, before discussing the methodologies in detail, it seems worthwhile to discuss some of the concepts that play a role in evaluating the structure of a program.

Structural analysis concepts

While most structural analysis concepts apply at more than one level of a software system, it is convenient in this discussion to separate them into three categories. Concepts are discussed that have their major impact at the code level, the module level, and the software system level.

Code-level concepts. Concepts having their major impact at this level include abstraction, communication, clarity, and control flow constructs.

Abstraction. Abstraction is defined as the consideration of a quality apart from a particular instance. In programming, the application of abstraction ranks as one of the most important advances that has occurred in the last 20 years. It is the basis for high-level languages, virtual machines, virtual I/O devices, data abstractions, plus both top-down and bottom-up design.

The whole concept of bottom-up design consists of building up layers of abstract machines that get more and more powerful until only one instruction is needed to solve the problem. While people usually stop far short of defining that one superpowerful instruction, they do significantly enhance the environment in which they have to program. Device drivers, operating system primitives, I/O routines, and user-defined macros are built on the concept of abstraction. All of them raise the level at which the programmer thinks and programs.[3]

Often, the objective is to abstract many of the complicated interactions that can occur when many users or user programs are sharing the same machine. In other instances, a virtual machine is created to hide the idiosyncrasies of a particular machine from the user so that the resulting program will be more portable.

When the modular programming era began in the 1960's, many people hoped that hundreds of reusable building-block programs could be abstracted and added to their programming libraries so that they could finally begin to "build on the work of others." Unfortunately, as Weinberg noted, "Program libraries are unique; everyone wants to put something in but no one wants to take anything out."[6]

Communication. A program communicates with both people and machines. The effect of comments can be profound. A ten-year-old program I've seen that has the comment "subtle" in it is still left alone at all costs. Although the person who wrote the program is long gone, he has left a legacy of problems that will last as long as the program.

Another program contained the comment, "They made me do it!" This comment was undoubtedly an apology for corrupting the structure of the program to provide an expedient fix to a pressing problem. Clearly, the program is a little harder to understand and modify now. This type of change is not unusual. It's the apology that's unusual.

In the long run, changes tend to obscure the structure of a program, thus making the processes of error correction and feature addition difficult and dangerous. A well-written and well-maintained program is meant to communicate its structure to the programmer as well as to give instructions to the machine. I believe that the life-cycle cost of operating a program usually depends far more on how well it communicates with people than on how fast it initially runs.

Clarity. It has been said that a person who writes English clearly can write a program clearly. In studying English, we are taught first to read and then to write. This seems to work well. In programming, however, we are usually taught only to write. I think we miss something by not learning to read programs first. At one time it was even considered fashionable to write unreadable programs. It got so bad that one language, famous for its "one-liners," was dubbed a "write-only" language.

The structure* of an article, paper, or book is very important in clearly communicating ideas. The structure of a program is equally important in communicating both the algorithm and the context of a problem solution. This structure should be apparent when one reads a program.

Clarity of program structure was obviously not the primary concern of the person who wrote the program represented in Figure 3. This program has been running for more than 10 years. Fortunately, few changes or feature enhancements have been required. In an attempt to understand how the program worked, this diagram was drawn by the last person who had to change it.

The "structuredness" of this program—and, for that matter, of any program—is not well-defined. There is still no generally accepted metric for characterizing the merit of a program structure. The unavailability of such a metric results in some strange phenomena. For example, do you know someone who writes complicated and unintelligible code, who spends long hours and late nights on it, finally getting it "done" just before the deadline—all the while letting everyone know how difficult his task is and what a hero he is for having gotten it done just in time?

In contrast, consider the neat, well-organized programmer who takes care to plan ahead, do a proper design, document her work, and get done well ahead of the deadline, with no one even aware that she was involved. How often have you thought "Boy, John has certainly earned his wings with that difficult program, while Jane hasn't had a chance to prove herself."

In the best of all worlds, the criterion of clarity could be applied quantitively. Lacking that, we'll have to stick with peer pressure applied in design reviews and code walkthroughs to ensure the clarity of the final product.

Control flow constructs. The concept of limiting the number and type of control flow constructs, to more clearly express algorithms, is now pretty generally accepted. The notation recommended by Michael Jackson[7] is shown in Figure 4.

The *sequence* and *selection* constructs shown in this figure can be generalized in the obvious way to a sequence or selection of *N* items for arbitrary *N*. The *iteration* stands for "zero or more" program executions. The advantage of

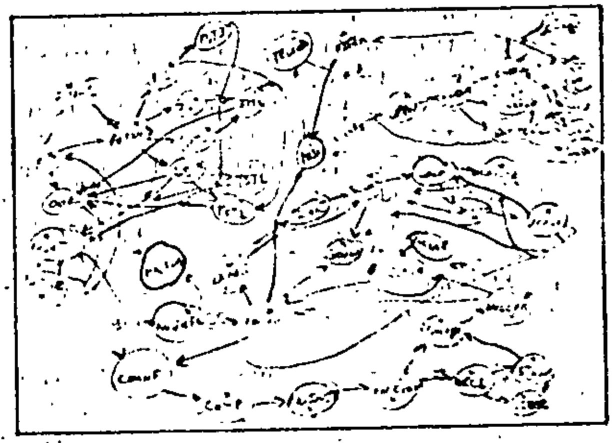


Figure 3. A clearly presented program structure?

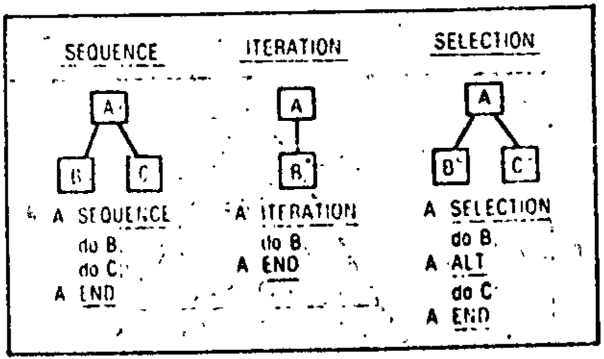


Figure 4. The three basic control flow constructs.

*An adequate but not inspired definition of "structure" is "the arrangement or interrelation of parts as dominated by the general character of the whole."

these constructs is that they tend to provide a map of the program structure rather than an itinerary of tasks.

The transformation of a program that uses these graphical constructs to structure text (also shown in Figure 4) is remarkably straightforward, as is the later transformation to a specific programming language.

Module-level concepts. Cohesion, coupling, complexity, correctness, and correspondence are concepts with major impact at the module level.

Cohesion. Cohesion is the "glue" that holds a module together. It can also be thought of as the type of association among the component elements of a module. Generally, one wants the highest level of cohesion possible. While no quantitative measure of cohesion exists, a qualitative set of levels for cohesion has been suggested by Constantine[8] and modified by Myers.[9] The levels proposed by Constantine are shown in Figure 5.

Coincidental cohesion is Constantine's lowest level of cohesion. Here, the component parts of a module are there only by coincidence. No significant relationship exists among them.

Logical cohesion is present when a module performs one of a set of logically related functions. An example would be a module composed of 10 different types of print routines. The routines do not work together or pass work to each other but logically perform the same function of printing.

Temporal cohesion is present when a module performs a set of functions related in time. An initialization module performs a set of operations at the beginning of a program. The only connection between these operations is that they are all performed at essentially the same time.

Procedural cohesion occurs when a module consists of functions related to the procedural processes in a program. Functions that can be well represented together on a flowchart are often grouped together in a module with procedural strength. Conversely, when a program is designed by using a flowchart, the resulting module often has procedural cohesion.

Communicational cohesion results when functions that operate on common data are grouped together. A data abstraction, or data cluster,[10] is a good example of a module with communicational cohesion.

Sequential cohesion often results when a module represents a portion of a data flow diagram. Typically, the modules so formed accept data from one module, modify or transform it, and then pass it on to another module.

Functional cohesion results when every function within the module contributes directly to performing one single function. The module often transforms a single input into a single output. An example often cited is square root. This is the highest level of cohesion in the hierarchy. As such, it is desirable whenever it can be achieved.

A program of any reasonable size will usually contain modules of several different levels of cohesion. Many modules simultaneously exhibit characteristics of a multiplicity of levels. Where possible, functional, sequential, and communicational strength modules should be given preference over modules with lower levels of cohesion. On a scale of 0 to 10, Yourdon[8] rates coincidental, logical, and temporal cohesion as 0, 1, and 3, respectively. Procedural cohesion would score 5. Communicational, sequential, and functional cohesion would score 7, 9, and 10, respectively.

While levels of cohesion can be useful guides in evaluating the structure of a program, they don't provide a clear-cut method for attaining high levels of cohesion. Furthermore, levels of cohesion do not allow us to say that program A is right and program B is wrong. They do, however, represent a definite step forward. Before levels of cohesion were introduced, there was no recognized basis for comparison. Now, at least one can say that structure A is probably better than structure B.

Coupling. Coupling is a measure of the strength of interconnection (i.e., the communication bandwidth) between modules. In Figure 6, two program structures are represented that would result in significantly different degrees of coupling.

High coupling among program modules results when a problem is partitioned in an arbitrary way such as cutting off sections of a flowchart. This method of chopping up a large program often complicates the total job because of the resultant tight coupling between the pieces. This latter type of partitioning leads to "mosaic" modularity.[11]

The other extreme in structuring a program is to consider only pure tree structures. These structures give rise to the concept of hierarchical modularity and provide many advantages for abstraction, testing, and later mod-

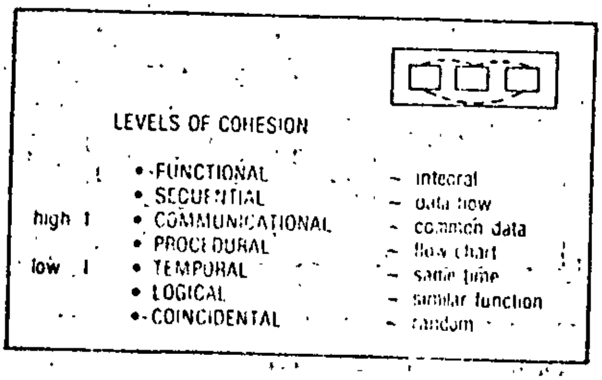


Figure 5. Levels of cohesion.

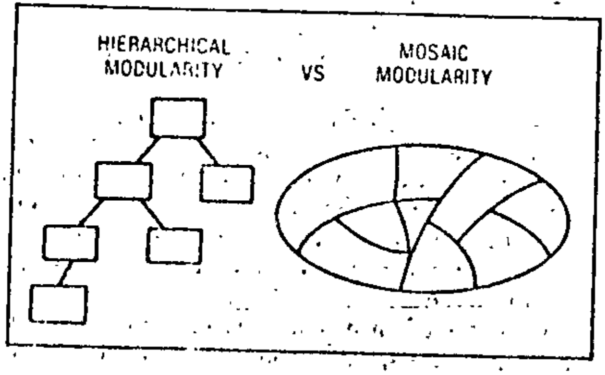


Figure 6. Partitioning method affects level of coupling.

ification. Jackson would accuse you of "arboricide" (the killing of trees) whenever you deviate from a pure hierarchical tree structure. Brooks has said, "I am persuaded that top-down design (incorporating hierarchy, modularity, and stepwise refinement) is the most important new programming formalization of the decade."[12] And Dijkstra has said that "the sooner we learn to limit ourselves to hierarchical program constructs the faster we will progress."[13]

Modular programs can be characterized as

- implementing a single independent function,
- performing a single logical task,
- having a single entry and exit point,
- being separately testable, and
- being entirely constructed of modules.

When these rules are followed, a set of nested modules result that can be connected in a hierarchy to form large programs. In an attitude survey,[14] users perceived that modular programs were easier to maintain and change, easier to test, and more reliable. The major perceived disadvantage was the feeling that the final program was less efficient than it could have been.

When modularity is used without hierarchy, one can only implement independent functions that can be executed in sequence, which corresponds to drawing circles around portions of a flowchart. Although this approach tends to work on small programs, it can seldom be applied to complex programs without seriously compromising module independence, connectivity, and testability. Only when the concepts of modular programming are combined with the concepts of hierarchical program structure can one implement arbitrarily complex functions and still maintain module integrity.

Modularity can be applied without hierarchy in cases that lend themselves naturally to the efficient use of a very high level language. Very high level language statements are examples of functions that can be implemented relatively independent of each other but still be strung together sequentially in a useful form. Unfortunately for most applications, the design of a convenient and efficient very high level language is difficult.

Hierarchical modularity forms an extremely attractive foundation for most of the other software engineering techniques. While some of these techniques can be used without having a hierarchical program structure, the primary benefit can only be gained when the techniques are used as a unit and build on each other. Specifically, a hierarchical modular program structure enhances top-down development, programming teams, modular programming, design walkthroughs, and other techniques that deal with improving the development process.

Complexity. The control of program complexity is the underlying objective of most of the software engineering techniques. The concept of "divide and conquer" is important as an answer to complexity, provided it is done correctly. When a program can be divided into two independent parts, complexity is reduced dramatically, as shown in Figure 7.

Consider program A, where you have access to only the input and the output.[9] A noble goal would be to completely test this program by executing each unique path. In the example shown, there are approximately 250 billion unique paths through this module. If you were capable of performing one test each millisecond, it would take you eight years to completely test all of the unique paths. If, however, you had knowledge of what was inside the program and recognized that it could be partitioned into two independent modules B and C, which have low connectivity and coupling, your testing job could be reduced. To test both of these modules separately requires that you only test the one million unique paths through each module. At one millisecond per test, these tests would take a total of only 17 minutes.

In this particular example, from a testing viewpoint, it is clearly worth trying to partition the problem so that small, independently testable modules can be dealt with instead of just the input and output of a large program. Unfortunately, partitioning most programs into independently testable modules requires much more work than simply drawing small circles around portions of the flowchart.

It should also be clear from this example that the testing problem is best solved during the design stage. It is impossible to exhaustively test any program of significant size. Testing is experimental evidence.[7] It does not verify correctness. It simply raises your confidence.

Correctness. A "correct" program is one that accurately implements the specification. A "correct" program often has limited value since the specifications are in error. Again, correctness cannot be verified by testing. Searching for errors is like searching for mermaids. Just because you haven't seen one doesn't mean they don't exist.

It is also unfortunate that, for most problems, mathematical proofs of correctness are as difficult to produce as a correct program. Some people can write and prove their program simultaneously. For most of us, however, day-to-day proving of programs is still a long way off.

The most promising approach for the near future may lie in finding a constructive proof of correctness. We will really have something if a design methodology can be

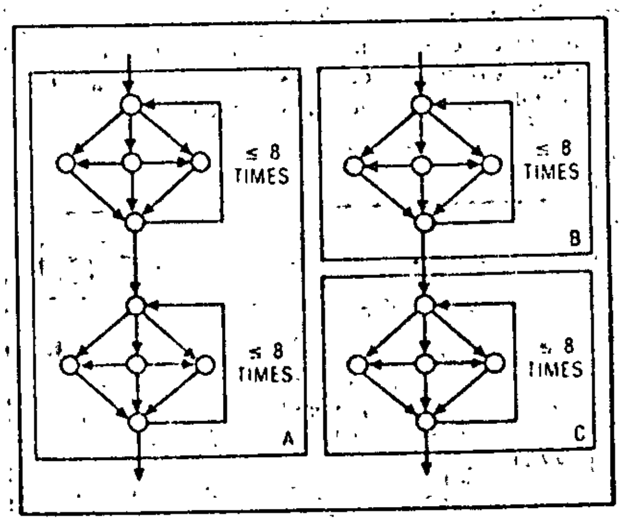


Figure 7. Control of complexity.

found that leads one through the design process step-by-step and guarantees the correctness of the final program if each of the steps has been done correctly. While I don't remember that 321 × 25 is 8025, I do remember that 5 × 1 = 5; and 5 × 2 = 10, and 5 × 3 = 15, etc. Knowing these values and the steps of multiplication, I can rest assured that 8025 is indeed the correct answer. If only a program design process existed that was as foolproof and easy to apply.

Without such a process, we must live with a limitless capacity for producing error. Weinberg once pointed out that errors can be produced in arbitrarily large numbers for an arbitrarily low cost.

Correspondence. In Jackson's view, perhaps the most critical factor in determining the life-cycle cost of a program is the degree to which it faithfully models the problem environment[7]—that is, the degree to which the program model corresponds to the real world. All too often, a small local change in the problem environment results in a large diffuse change in the program.

The world is always bigger than the program specification says it is, but a specification can always be extended if it corresponds to reality. Since users tend to be gradualists, the changes in a realistic problem model will tend to be gradual. If the program structure is formed around the static instead of the dynamic properties of the problem, it should prove to be more resilient to changes.

While a program's model of the world cannot be complete, it must at least be useful and true. If these criteria are met, many maintenance and feature enhancement problems will be avoided in the future.

System-level concepts. Concepts of major impact at this level include consistency, connectivity, continuity, change, chaos, optimization and packaging.

Consistency. An important objective of a good design methodology is that it should produce a consistent program structure independent of whoever is applying it. Three different programs—created by using the same design methodology to model the same problem environment—should have the same basic structure. Unless consistent designs can be achieved, there can never be a true right or wrong structure for a given problem solution.

One problem with most design methodologies is that there is no one consistently obtainable solution. Instead, designers seem to pull unique solutions out of the air. Consequently, there is never discussion of a solution being right or wrong—only discussion of my style versus your style.

Connectivity. The harmful effects of high connectivity on system modifiability can best be illustrated by using an analogy.[15]

Consider a system composed of 100 light bulbs. Each light in the system can either be on or off. Connections are made between the light bulbs so that if the light is on, it has a 50 percent chance of going off in the next second. If the light bulb is off, it has a 50 percent chance of going on in the next second—provided one of the lights to which it is connected is on. If none of the lights connected to it is on, the light stays off. Sooner or later, this system of light bulbs will reach an equilibrium state in which all of the lights go off and stay off.

The average length of time required for this system to reach equilibrium is solely a function of the interconnection pattern of the lights. In the most trivial interconnection pattern, all of the lights operate independently. None of them is connected to any of its neighbors. Here, the average time for the system to reach equilibrium is approximately the time required for any given light to go off—about two seconds. Thus, the system can be expected to reach equilibrium in a matter of seconds.

At the other extreme, consider the case when each light in the array is fully connected to all other lights in the array—that is, assume that there is a connectivity matrix for the lights similar to the program connectivity matrix shown on the left side of Figure 8, where *N* is equal to 100. The array on the left side of the figure then describes the connectivity matrix of the lights and shows that every light is connected to every other light. In this case, the length of time required for the system to reach equilibrium is 10^{22} years. This is a very long time when you consider that the current age of the universe is only 10^{10} years.

Now, consider one final interconnection pattern in which the set of 100 lights is partitioned into 10 sets of 10 lights each, with no connections between the sets, but with full interconnection within each set. In this case, the time required for the system of lights to reach equilibrium is about 17 minutes. This example dramatically shows the effect of connectivity. In terms of the concepts presented earlier, this example corresponds to high cohesion within each module and low coupling between modules.

Much as proper physical partitioning can dramatically reduce the time required for the system of lights to reach equilibrium, proper functional partitioning can dramati-

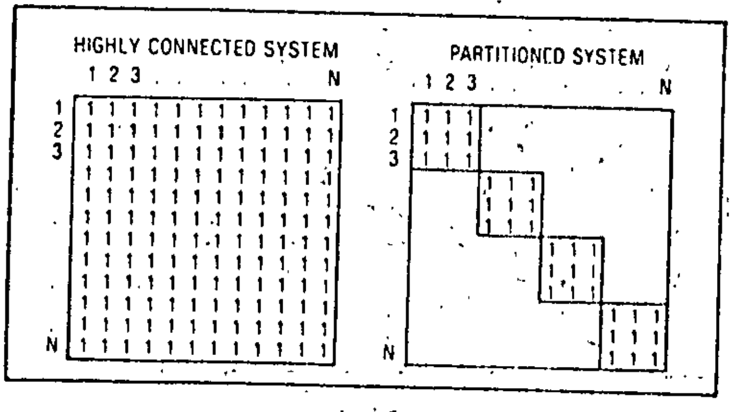


Figure 8. Low connectivity implies low maintenance costs.

LAW OF CONTINUING CHANGE: A SYSTEM THAT IS USED UNDERGOES CONTINUING CHANGE UNTIL IT IS JUDGED MORE COST-EFFECTIVE TO FREEZE AND RECREATE IT.

LAW OF INCREASING UNSTRUCTUREDNESS: THE ENTROPY (DISORDER) OF A SYSTEM INCREASES WITH TIME UNLESS SPECIFIC WORK IS EXECUTED TO MAINTAIN OR REDUCE IT.

Figure 9. Continuity/change/chaos.

cally reduce the time required for a program that is being debugged to reach stability.

Continuity/change/chaos. As noted by Belady and Lehman,[16] a large program often seems to live a life of its own, independent of the noble intentions of those trying to control it. Two important observations are summarized in the Law of Continuing Change and the Law of Increasing Unstructuredness shown in Figure 9.

These laws dramatize the key role played by the program structure during the life cycle of software systems. The natural order of things is to produce disorder. If the program structure is unclear from the beginning, things will only get worse later. These two laws coupled with a poor program structure have produced many of the maintenance-cost horror stories.

Optimization and packaging. All too often, people confuse packaging and design. Design is the process of partitioning a problem and its solution into significant pieces. Optimization and packaging consist of clustering pieces of a problem solution into computer load modules that run within system space and time requirements without unduly compromising the integrity of the original design.[8]

At least three different types of modules must be considered in programming—functional modules, data modules, and physical modules. Packaging is concerned with placing functional modules and data modules into physical modules. In packaging a program, several of these pieces of the program may be put together as one load module or may even be written together as one program.

It is in the packaging phase of a design that optimization should be considered for the first time. This phase is done at the end, and great care should be taken to preserve the program structure that you have worked so hard to create. In Jackson's words, "It is easy to make a program that is right, faster. It is difficult to make a program that is fast, right." Once an optimization has been cast in code, it's like concrete. It is very difficult to undo.

Functional decomposition

Functional decomposition is simply the divide-and-conquer technique applied to programming, as shown in Figure 10. Various forms of functional decomposition have been popularized by a host of people including Dijkstra, Wirth, Parnas, Liskov, Mills, and Baker.[17-22]

By viewing the stepwise decomposition of the problem and the simultaneous development and refinement of the program as a gradual progression to levels of greater and greater detail, we can characterize functional decomposition as a *top-down* approach to problem-solving. Conversely, we can form and layer groups of instruction sequences together into "action clusters," starting at the atomic machine instruction level and working our way up to the complete solution. This approach leads to a *bottom-up* method.[21]

Often, the preferred strategy is to shift back and forth between top-down functional decomposition and the bottom-up definition of a virtual machine environment.

Design strategy. The design process can be divided into the following steps:[19]

(1) Clearly state the intended function.
(2) Divide, connect, and check the intended function by reexpressing it as an equivalent structure of properly connected subfunctions, each solving part of the problem.
(3) Divide, connect, and check each subfunction far enough to feel comfortable.

In following this procedure, the key to successful program design is rewriting followed by more rewriting. Every effort should be made at each step to conceive and evaluate alternate designs.

A useful mind set is to pretend that you are programming on a machine that has a language powerful enouch to solve your problem in only a handful of commands. In your level 1 decomposition, you simply write down that handful of commands and you have a complete program. In your level 2 decomposition, you try to refine each of your level 1 instructions into a set of less powerful instructions. By continuing to successively refine each instruction, one level at a time, you eventually get to a program that can be executed on your own real computer. In carrying out this process, you will have decomposed the problem into its constituent functions by "stepwise refinement."

There are several problems involved in applying this technique. First, the method specifies that a functional decomposition be performed, but it does not say what you are decomposing with respect to. One can decompose with respect to time order, data flow, logical groupings, access to a common resource, control flow, or some other criterion. If you decompose with respect to time, you get modules like initialize, process, and terminate, and you have a structure with temporal cohesion. If you cluster functions that access a shared data base, you have made a start toward defining abstract data types and will get communicational cohesion. If you decompose using a data flow chart, you may end up with sequential cohesion. If you decompose around a flowchart, you will often end up with logical cohesion. The choice of "what to decompose with respect to" has a major effect on the "goodness" of the resulting program and is therefore the subject of much controversy.

The major advantage of functional decomposition is its general applicability. It has also been used by more people longer than any of the other methods discussed. The dis-

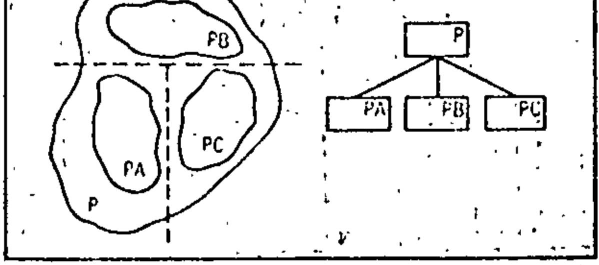


Figure 10. Functional decomposition.

advantages are its unpredictability and variability. The chance of two programmers independently solving a given problem in the same way are practically nil. Thus, each new person exposed to a program starts by saying, "This isn't the way I would have done it, but. . ." To some extent, this problem can be reduced by using functional decomposition in combination with some other technique that determines what each function should be composed with respect to.

McDonald's example. The McDonald's functional decomposition solution, presented below, is patterned (with permission) after a story called "Getting it Wrong" that has been related by Michael Jackson on numerous occasions in his short courses and seminars. The McDonald's frozen-food warehouse problem is, of course, entirely fictitious.

Problem specification. McDonald's frozen-food warehouse receives and distributes food items. Each shipment received or distributed is recorded on a punched card that contains the name of the item, the type of shipment (R for received, D for distributed), and the quantity of each item affected. These transaction cards are sorted by another program and appear grouped in alphabetical order by item name. A management report, showing the net change in inventory of each item, is produced once a week. The input file and output report formats are shown in Figure 11.

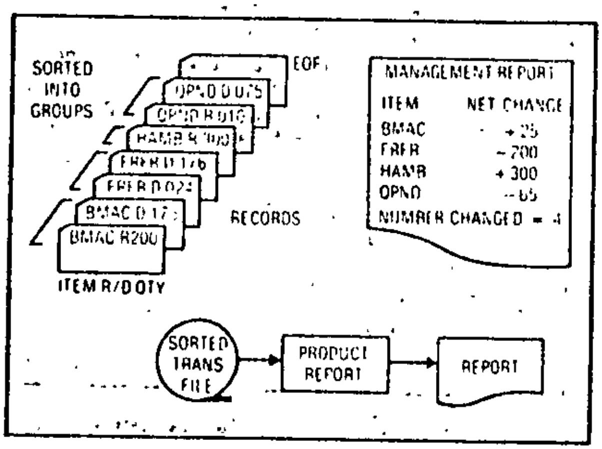


Figure 11. McDonald's warehouse I/O formats.

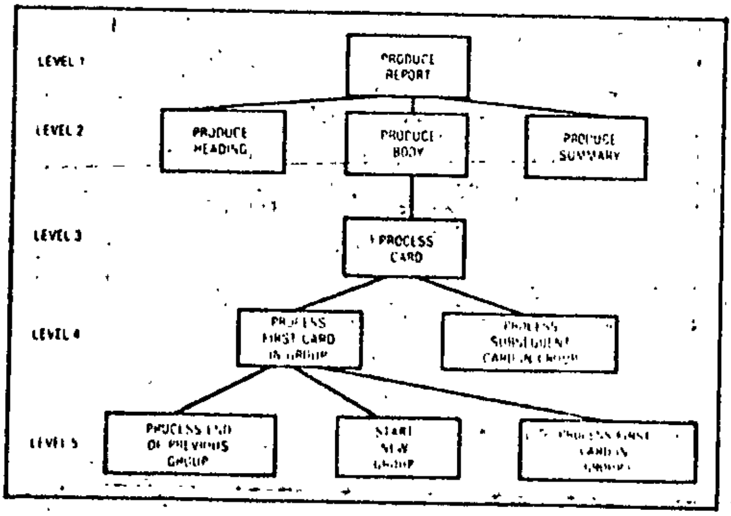


Figure 12. A five-level functional decomposition.

Design phase. The hero who originally designed this program is named Ivan. Ivan is a very "with it" fellow. He swore off GO-TOs years ago. His code is structured like the Eiffel Tower. He can whip out a neatly indented structured program in nothing flat. In doing his design, Ivan was careful to do the five-level, hierarchical, functional decomposition shown in Figure 12.

In this figure, PRODUCE REPORT is shown to be a sequence of PRODUCE HEADING followed by PRODUCE BODY followed by PRODUCE SUMMARY. PRODUCE-BODY is shown to be an iteration of PRODUCE CARD that is a selection between PROCESS FIRST CARD IN GROUP and PROCESS SUBSEQUENT CARD IN GROUP.

Clearly, recognizing the first card of each item group is important. Once this card is found, everything else falls into place.

At this point, it may be worth examining the structure of Ivan's program. The obvious question is, "Is this a good decomposition?" The obvious reply is, "Good with respect to what?"

If we apply the concept of cohesion to this structure, it might seem that the level 1 PRODUCE REPORT module has temporal cohesion since the PRODUCE HEADING module is something like an initialization module and the PRODUCE SUMMARY module is something like a terminate module. On the other hand, one could argue that the heading, body, and summary are such integral parts of the report that this is really functional cohesion.

Likewise, the PROCESS CARD module seems to have been partitioned along temporal lines as well. On the other hand, PROCESS CARD seems to be an integral part of the PRODUCE BODY module, so maybe PRODUCE BODY is also functionally cohesive.

As you can tell by the preceding examination, while levels of cohesion may constitute an improvement over having no basis for comparison, they are still difficult to apply consistently. In cases where more than one type of cohesion seem to be present, the rule[8] is to assume that it is really the higher of the two levels.

Ivan wrote the level 1, 2, and 3 programs shown in Figure 13. The level 4 decomposition shown in Figure 14 started to look like a finished product. The level 5 decomposition shown in Figure 15 was the finished product.

In his normal thorough manner, Ivan volume-tested his program and turned it over to the user. It was perfect except for one small glitch. You see, the systems programmers were still playing with the compiler and obviously hadn't fixed all the errors. As a result, the program worked fine, but some garbage appeared on the first line of the output immediately after the headings. It was believed, of course, that this would disappear as soon as they fixed that %&***? > compiler.

Friendly user phase. Now, Ronald McUser is a friendly sort of person. Since he was in a hurry to use the program, he did not complain about the small glitch that would, of course, disappear very soon. Ronald's boss, Big Mac, however, failed to see the humor of it all. The compiler had been fixed for three months now, and the management report that went to the board of directors still had that garbage in it.

Finally, one day, Ronald could stand it no longer. The program *had* to be fixed. When Ronald arrived, Ivan was in his cubicle, listening to an audio cassette of Dijkstra's Turing lecture. He, of course, didn't learn anything; that's the way he had always done his designs. Ronald showed Ivan a printout.

After a few MMMs and AAAHHHHs and AAAHHHAAAs, he saw the problem. This first time through the program, there was no previous group. Thus, the output was just random data. The solution to any first-time-through problem is, of course, obvious. Add a first-time switch (see Figure 16).

Maintenance phase. Six months later, our hero was in McDonald's "think room" when Ronald McUser came in and said, "I put 80 transactions in last week and nothing came out!" Our hero looked at the printout and saw that, indeed, only the heading had come out.

Ivan knew immediately what the problem was. It must be a hardware problem. After all, his program had been running nearly a year now with only one small complaint. Ivan spent most of the night running hardware diagnostics until Steve Saintly, a keypunch operator, wandered by and said, "That nationwide special on Big Macs last week was all we handled. Everything else had to wait."

A little later, Sally Saintly came by and said, "Isn't it about time you included those new Zebra sodas in the management report?"

By this time, Ivan was very discouraged with his diagnostics, so he followed last week's inputs through the code. "Horrors! There was only one item group processed last week—Big Macs!"

As Ivan soon discovered, the last item group was never processed. Since only one item group was processed all week, nothing was output. Up until this time, only Zebra sodas had been skipped. Since they were not a big winner, it seems that no one had even cared that they had been left off. In fact, everyone assumed they were being left off on purpose. Ivan's solution is shown in Figure 17.

```
LEVEL 1     P: PRODUCE REPORT:
               STOP;

LEVEL 2     P: PRODUCE HEADING.
               PRODUCE BODY:
               PRODUCE SUMMARY:
               STOP;

LEVEL 3     P: PRODUCE HEADING.
                 READ STF.
                 DO WHILE (NOT EOF-STF):
                       PROCESS CARD.
                       READ STF.
                 END:
               PRODUCE SUMMARY.
               STOP.
```

Figure 13. Steps in functional decomposition.

```
P: PRODUCE HEADING:

     READ STF:
     DO WHILE (NOT EOF —STF):
           IF FIRST CARD IN GROUP THEN
              PROCESS FIRST CARD IN GROUP:
              ELSE PROCESS SUBSEQUENT CARD IN GROUP.
           READ STF.
     END:

   PRODUCE SUMMARY:
   STOP.
```

Figure 14. Level 4 functional decomposition.

```
P  PRODUCE HEADING:

     READSTF.
     DO WHILE (NOT EOF—STF):
        IF FIRST CARD IN GROUP THEN
           DO.
                  PROCESS END OF PREVIOUS GROUP:

                  PROCESS START OF NEW GROUP:
                  PROCESS CARD:

           END.
        ELSE DO: PROCESS CARD:
              END:
        READ STF:
     END.

   PRODUCE SUMMARY:
   STOP:
```

Figure 15. The final functional decomposition.

```
P: PRODUCE HEADING:
   SW1 = 0
     READSTF:
     DO WHILE (NOT EOF — STF):
        IF FIRST CARD IN GROUP THEN
           DO. IF SW1 = 1 THEN
               DO. PROCESS END OF PREVIOUS CROUP:
               END. SW1 = 1
                     PROCESS START OF NEW GROUP:
                     PROCESS CARD:

           END:
        ELSE DO: PROCESS CARD.
              END:
        READ STF:
      END.

   PRODUCE SUMMARY.
   STOP.
```

Figure 16. Quick fix no. 1.

Meanwhile, Sally Saintly asked; "Why didn't you see that during all the volume-testing you did? You tied up the machine for most of a day."

The answer again is obvious. In volume-testing, you put in thousands of inputs but don't look at the output.

Passing the baton. Six months later, Ivan was feeling pretty pleased with himself. He had just turned the program over to a new hire. There would still be some training, but everything should go well. After all, hadn't the program run for nearly a year and a half with only a couple of small problems? Suddenly, Ronald burst in, "I thought you fixed this first-line problem. Here it is again."

Ivan knew immediately what the problem was. The new program librarian they had forced him to use had put in an old version of the program without his first patch.

```
P: PRODUCE HEADING;
   SW1: = 0;
      READSTF;
      DO WHILE (NOT EOF—STF);
         IF FIRST CARD IN GROUP THEN
            DO; IF SW1 = 1 THEN
                DO; PROCESS END OF PREVIOUS GROUP;
                END; SW1: = 1,
                     PROCESS START OF NEW GROUP,
                     PROCESS CARD;

            END;
         ELSE DO; PROCESS CARD;
               END;
         READ STF;
      END;
                 PROCESS END OF LAST GROUP,
   PRODUCE SUMMARY,
   STOP;
```

Figure 17. Quick fix no. 2.

After many heated comments plus a core dump, Ivan was still baffled. Finally, in desperation, he sat down to look at the input data and found out there wasn't any: Last week, a trucker's strike had shut down the warehouse. Nothing came in; nothing went out. They ran the program anyway. Good grief, who would have thought that they would run the program with no inputs!

The problem, as it turns out, was that the new PROCESS END OF LAST GROUP module needed protection just like the PROCESS END OF PREVIOUS GROUP module had before. Since that first-time switch had worked so nicely earlier, it was clearly the solution to apply again (see Figure 18).

We all know that Ivan's troubles are over now. Or are they? Two months later, Sally Saintly came in and said, "Where are the Zippo sandwiches? They were in for two months, but now they've suddenly disappeared from the report."

After complaining that the new hire was supposed to be maintaining that program now, Ivan looked at the input data and noticed that only one order per item had been issued during the whole run.

"What happened?" he exclaimed.

It seems that a new manager, Mary Starr, had started a new policy to try to get things better organized. She had asked each of the stores to place only one order a day instead of placing orders at random. She had also said that it would be nice if they could schedule things so that the warehouse had to be concerned only with receiving one particular item on one day and with distributing that item the next day. In addition, she wanted the management report program run once a day from now on. The effect on Ivan's program was that Zippo sandwiches was dropped.

Instead of moving the set for SW2, the safest thing to do—according to the principles of defensive programming—is to add an extra set. Since you don't know what you're doing, you never touch a previous fix—just add a new one (see Figure 19).

Now we can all rest assured that Ivan's program works, right?

```
P: PRODUCE HEADING;
   SW1: = 0;   SW2 = 0.
      READSTF;
      DO WHILE (NOT EOF—STF);
         IF FIRST CARD IN GROUP THEN
           DO; IF SW1 = 1 THEN
                DO; PROCESS END OF PREVIOUS GROUP;
                END, SW1: = 1;
                     PROCESS START OF NEW GROUP;
                     PROCESS CARD;

           END;
         ELSE DO; PROCESS CARD; SW2 = 1;
               END;
         READ STF,
     END; IF SW2 = 1 THEN
           DO, PROCESS END OF LAST GROUP,
           END;
  PRODUCE SUMMARY;
  STOP;
```

Figure 18. Quick fix no. 3

```
P: PRODUCE HEADING;
   SW1. = 0; SW2: = 0;
      READSTF;
     [DO WHILE (NOT EOF—STF);
         IF FIRST CARD IN GROUP THEN
           [DO; IF SW1 = 1 THEN
                [DO;   PROCESS END OF PREVIOUS GROUP;
                [END, SW1 = 1;
                       PROCESS START OF NEW GROUP;
                       PROCESS CARD;
                       SW2 = 1,

           [END;
         ELSE DO  PROCESS CARD; SW2: = 1;
               [END;
         READ STF.
     [END. IF SW2 = 1 THEN
           [DO. PROCESS END OF LAST GROUP;
           [END;
  PRODUCE SUMMARY;
  STOP;
```

Figure 19. Quick fix no. 4.

The effect of all of these changes on the program structure is shown in Figure 20.

What was Ivan's major sin? Simply that the program structure didn't correspond to the problem structure. In the original data, there existed something called an item group. There is no single component in Ivan's program structure that corresponds to an item group. Thus, the actions that should be performed once per group, before a group, or after a group have no natural home. They are spread all over the program, and we have to rely on first-time switches and the like to control them. Instead of components that start a new group, process a group, or process the end of a group, we have a mess.

In Jackson's words, when this correspondence is not present, the program is not poor, suboptimal, inefficient, or tricky. It's *wrong*. The problems we have seen are, in reality, nothing but self-inflicted wounds stemming from an incorrect program structure.

Data flow design

The data flow design method first proposed by Larry Constantine[8] has been advocated and extended by Ed Yourdon[8] and Glen Myers.[9] It has been called by several different names, including "transform-centered design" and "composite design." In its simplest form, it is nothing more than functional decomposition with respect to data flow. Each block of the structure chart is obtained by successive application of the engineering definition of a black box that transforms an input data stream into an output data stream. When these transforms are linked together appropriately, the computational process can be modeled and implemented much like an assembly line that merges streams of input parts and outputs streams of final products.[8,9]

Design strategy. The first step in using the data flow design method is to draw a data flow graph (see Figure 21). This graph is a model of the problem environment that is transformed into the program structure. An example of its use will be given later.

While the modules in functional decomposition often tend to be attached by a "uses" relationship, the bubbles in a data flow graph could be labeled "becomes." That is, data input A "becomes" data output B. Data B becomes C, C becomes D, etc. The only shortcoming of this decomposition is that it tends to produce a network of programs—not a hierarchy of programs. Yourdon and Constantine solve this problem by simply picking the data flow graph up in the middle and letting the input and output data streams "hang down" from the middle. At each level, a module in one of the input or output data streams can be factored into a "get" module, a "transform" module, and a "put" module. By appropriate linking, a hierarchy is formed. Thus, once the data flow graph is drawn, the hierarchical program structure chart can be derived in a relatively mechanical way.

Given the data flow graph of Figure 21, the modules of the structure chart are defined as GET A, GET B, and GET C. Also defined are the modules that transform A into B, B into C, C into D, and so on. The output module is illustrated by the PUT D module. The GET modules are called "afferent modules" and the PUT modules are called "efferent modules."[8] The TRANSFORM C TO D module is known as the "central transform."

The data flow design method can be broken into the following four basic steps:

(1) Model the program as a data flow graph.
(2) Identify afferent, efferent, and central transform elements.
(3) Factor the afferent, efferent, and central transform branches to form a hierarchical program structure.
(4) Refine and optimize.

This procedure is represented schematically in Figure 22.

Note that while the connections between modules in the data flow graph context were motivated by a "becomes" or "consumes/produces" relationship, the factoring procedure leads to modules that are connected by a "calls/is called by" relationship. Thus, the hierarchy formed is really being artificially imposed by the scheduling and has little to do with modeling the problem in

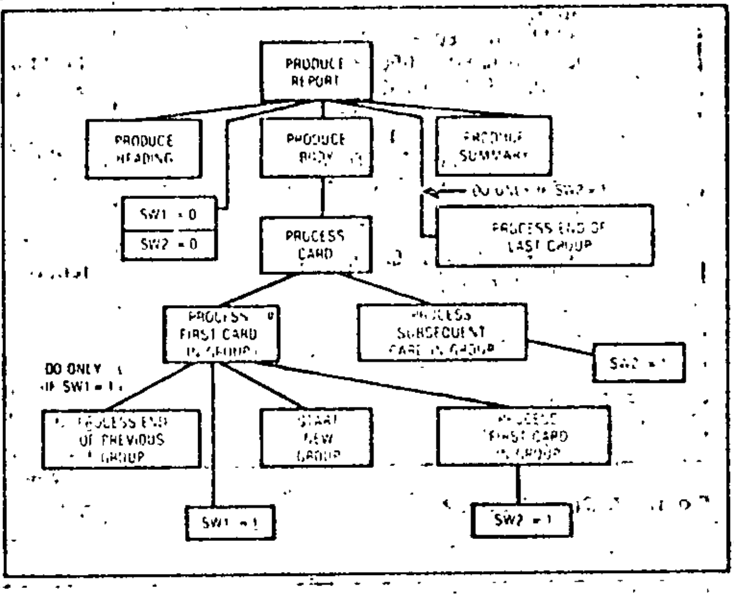


Figure 20. Functional decomposition with quick fixes.

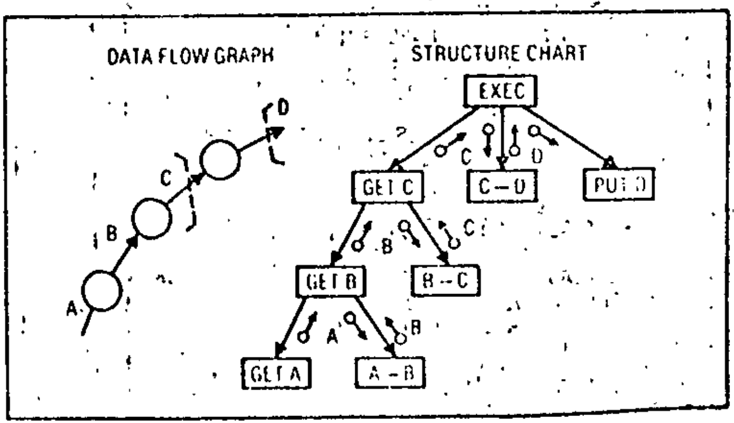


Figure 21. Data flow design method.

hierarchical fashion. This contrasts with both the "uses" relationship of decomposition with respect to function and the "is composed of" relationship that motivates a data structure design.

Also note that a lot of data passes between modules in the structure in assembly-line fashion. This results in sequential cohesion. By Constantine's measure of goodness, the data flow design method produces a very good program structure.

The act of concentrating the I/O functions in the afferent and efferent "ears" of the program structure may or may not produce a structure that models the problem environment accurately. This partitioning often seems artificial to me and would seem to violate the principle of correspondence.

The central transform is located between the data flow graph's most abstract input and most abstract output. While the graph shows it as simply transforming Cs into Ds, it often requires a sophisticated functional decomposition in its own right. That is, except for the "ears" which come from the data flow graph, one is forced right back to the art of functional decomposition.

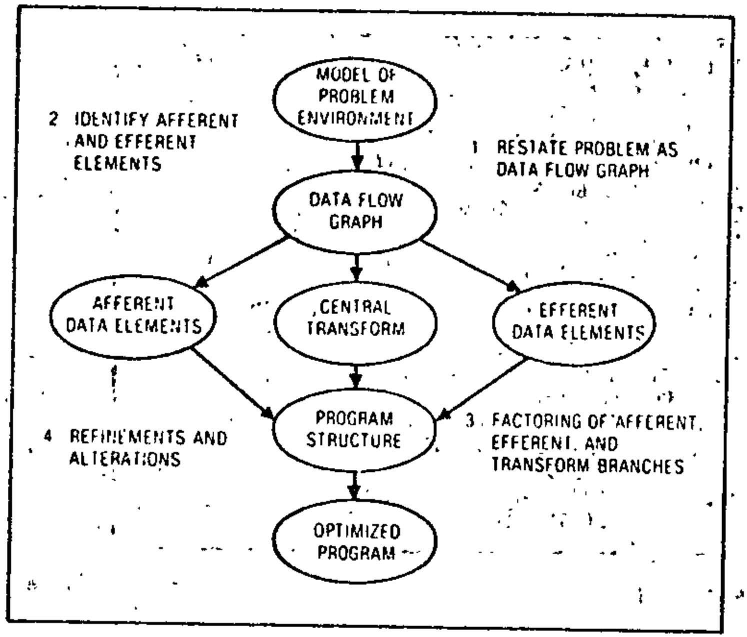

MODEL OF PROBLEM ENVIRONMENT
1 RESTATE PROBLEM AS DATA FLOW GRAPH
2 IDENTIFY AFFERENT AND EFFERENT ELEMENTS
DATA FLOW GRAPH
AFFERENT DATA ELEMENTS
CENTRAL TRANSFORM
EFFERENT DATA ELEMENTS
PROGRAM STRUCTURE
3 FACTORING OF AFFERENT, EFFERENT, AND TRANSFORM BRANCHES
4 REFINEMENTS AND ALTERATIONS
OPTIMIZED PROGRAM

Figure 22. Data flow design procedure.

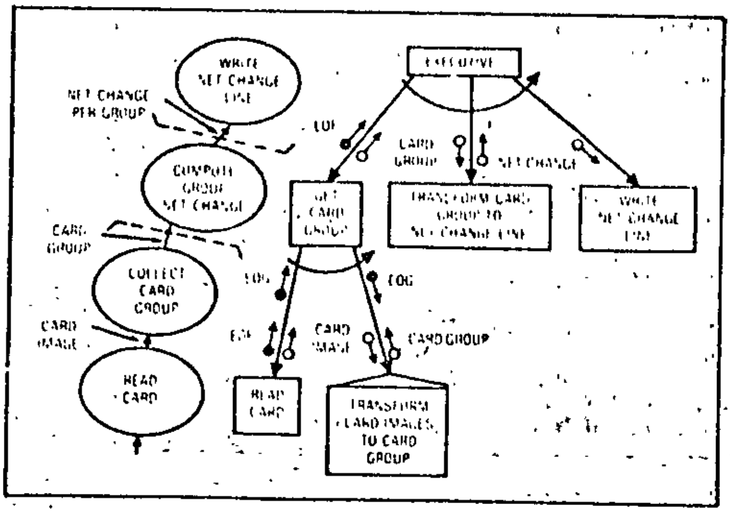


Figure 23. Data flow design structure.

McDonald's example. The heroine who designed the next program was Ivan's sister, Ivy. Ivy is a child of the late 1960's. When modular programming came in, she jumped right on the bandwagon. Her modules had only one entrance and one exit. She passed all her parameters in each call statement. Each of her modules performed only a single logical task, was independent, and could be separately tested. She read daily from the gospel according to Harlan Mills[19] and remembered nearly every error that she had ever made. She well remembers the day she designed this program.

Model problem. The data flow graph for the McDonald's example is shown on the left side of Figure 23. The data items being passed between bubbles are labeled, and the modules are named with an action verb and an object. Note that one or more *card images* "become" a *card group* after being processed by the COLLECT CARD GROUP function.

Afferent, efferent, and central transform elements. The most abstract input in the data flow graph is a *card group*. The most abstract output is the *net change* resulting from processing each card group. Thus, the COMPUTE GROUP NET CHANGE module is the central transform, the WRITE NET CHANGE LINE module is the efferent (or output) element, and the READ CARD and COLLECT CARD GROUP modules are the afferent (or input) elements.

Factor branches. Note that the concept of a card group emerges naturally out of the data flow diagram and leads to a reasonably straightforward program structure (see Figure 23). The TRANSFORM CARD IMAGES TO CARD GROUP module is shown with a roof that denotes lexical inclusion. That is, while TRANSFORM CARD IMAGES TO CARD GROUP is a functional module, it is not necessarily a physical module. In this example, it will be packaged together with the GET CARD GROUP module.

The curved arrows in Figure 23 denote iteration. That is, the GET CARD GROUP module calls its two subtending modules once per card image. The EXECUTIVE module calls its three subtending modules once per card group. Note that the READ CARD module is assumed to pass an "end of file" flag up the chain when it is detected. (The passing of control information is denoted by the small arrows with solid tails, while the passing of data is shown by the small arrows with open tails.) Ivy's program listing is shown in Figure 24.

Only three of the six modules are shown. The TRANSFORM CARD IMAGES TO CARD GROUP module was lexically included in the GET CARD GROUP module. The other two modules are not necessary for this particular discussion. Note that the READ CARD module leaves much to be desired.

Program testing. Ivy has learned to make good use of a program librarian. She wrote the program, gave the coding sheets to the librarian, and went back to her reading. The program librarian faithfully executed his duties and brought back a printout that said "CRITEM undefined."

Ivy had forgotten to make sure that all of her variables were initialized. Oh well, this should be easy. How about setting CRITEM: = XXXX at the beginning of the GET CARD GROUP module? That would be before CRITEM was used the first time, and when it has a symmetry with the way EOG (end of group) is initialized.

Unfortunately, something just didn't seem right about it. GET CARD GROUP is executed many times during the program, and CRITEM only needs to be initialized once.

Ivy decided to initialize CRITEM at the beginning, instead. While that means passing it as a parameter up two levels, it certainly sounded better than a bunch of first-time switches.

On the next run, Ivy's efforts were rewarded with some output—nothing fancy, but still some output.

"What happened to the heading?" exclaimed Ivy. "That data link must be dropping bits again."

"Where did you write out the heading?" asked the program librarian. Enter quick fix number 2, shown in Figure 25.

"Wait, what about that garbage in the front?"

The problem, of course, is that the program has no way of knowing when it's through with a group until it's already started processing the next group. Thus, the first card of a new group serves as a key to tell the rest of the program to send on the previous group. It can't do this, however, without distorting the structure.

"I think it's time to call out my secret weapon," said Ivy, "my UNREAD command." Enter quick fix number 3, shown in Figure 26.

Field debugging. This fix apparently worked fine for about two months. It was not exactly speedy, but it did work. Then, all of a sudden, a visit from on high. Big Mac himself came down and said, "Our people in Provo, Utah, have been trying to bring up your program, and it just doesn't work."

It turns out that in Provo they never found it necessary to buy a tape reader. Ivy's UNREAD operation didn't work on cards. That meant that Ivy had to find another way of UNREADing.

In this data flow design, the equivalent of an UNREAD is, at best, messy. It corrupts the structure badly, no matter how it's done. Modules end up storing internal states or values, and first-time switches abound.

In Ivy's case, she chose to read ahead by one, passing state information by SWT and storing the NEW CARD value within module READ CARD. Other solutions are possible, of course, but it isn't clear that they are a whole lot better (see Figure 27). The effect of these changes on the program structure is shown dramatically in Figure 28.

Ivy has committed arboricide, to use Jackson's words. What was a nice clean tree structure now has two programs calling the same READ CARD module. I think

```
EXECUTIVE: EOF: = FALSE;

  DO WHILE(EOF = FALSE).
     GET__CARD__GROUP(CG.EOF          );
     TRANSFORM__CG__TO__NC__LINE(CG,NC);
     WRITE__NC__LINE (NC.EOF);
  END;                   STOP;

GET__CARD__GROUP(CG.EOF                  ): EOG: = FALSE; I: = 0;
  DO WHILE (EOG = FALSE); I: = I + 1;
     READ__CARD(CI.EOG EOF                  );CG(I) = CI;
  END.                 RETURN.

READ__CARD(CI.EOG.EOF              ):
  NEW__CARD = READ STF;
     IF NEW__CARD__ITEM ≠ CRITEM THEN
        DO; EOG: = TRUE. CRITEM = NEW__CARD__ITEM;
                 END;
     ELSE     CI: = NEW__CARD.
     IF CI = EOF__STF THEN EOF: = TRUE: RETURN;
```

Figure 24. Data flow design program.

```
EXECUTIVE:   EOF: = FALSE.
   CRITEM = XXXX
  DO WHILE(EOF = FALSE):
     GET__CARD__GROUP(CG.EOF CRITEM      );
     TRANSFORM__CG__TO__NC__LINE(CG,NC);
     WRITE__NC__LINE (NC.EOF).
  END;                         STOP;

GET__CARD__GROUP(CG EOF CRITEM          ). EOG: = FALSE; I: = 0;
  DO WHILE (EOG = FALSE), I: = I + 1;
     READ__CARD (CI.EOG.EOF CRITEM          );CG(I) = CI;
  END;               RETURN;

READ__CARD(CI.EOG.EOF CRITEM           ):
  NEW__CARD. = READ STF;
     IF NEW__CARD__ITEM  ≠ CRITEM THEN
        DO; EOG = TRUE. CRITEM  = NEW__CARD__ITEM;
                END;
     ELSE    CI: = NEW__CARD;
     IF CI = EOF__STF THEN EOF: = TRUE: RETURN;
```

Figure 25. Quick fix no. 1.

```
EXECUTIVE:   EOF: = FALSE. WRITE HEADING.
   CRITEM: = XXXX
  DO WHILE(EOF = FALSE)
     GET__CARD__GROUP(CG EOF.CRITEM      ):
     TRANSFORM__CG__TO__NC__LINE(CG NC).
     WRITE__NC__LINE (NC.EOF).
  END;                         STOP;

GET__CARD__GROUP(CG.EOF CRITEM          ): EOG: = FALSE; I: = 0.
  DO WHILE (EOG = FALSE). I = I + 1.
     READ__CARD (CI.EOG EOF.CRITEM          ): CG(I) = CI;
  END;               RETURN;

READ__CARD(CI.EOG.EOF CRITEM           ).
  NEW__CARD = READ STF.
     IF NEW__CARD__ITEM  ≠ CRITEM THEN
        DO. EOG. = TRUE  CRITEM  = NEW__CARD__ITEM;
        UNREAD STF  END.
     ELSE    CI = NEW__CARD
     IF CI = EOF__STF THEN EOF: = TRUE  RETURN.
```

Figure 26. Quick fixes no. 2 and no. 3.

```
EXECUTIVE:   EOF: = FALSE.WRITE HEADING: SW1 = FALSE.
    CRITEM = XXXX,      READ  CARD(CI EOG EOF,CRITEM,SW1).
    DO WHILE(EOF = FALSE):
        GET__CARD__GROUP(CG.EOF,CRITEM SW1);
        TRANSFORM__CG__TO__NC__LINE(CG,NC);
        WRITE__NC__LINE(NC.EOF);
    END;                     STOP;

GET__CARD__GROUP(CG.EOF CRITEM SW1). EOG: = FALSE; I: = 0;
        DO WHILE (EOG = FALSE). I: = I + 1;
        READ__CARD (CI.EOG.EOF,CRITEM.SW1):CG(I) = CI;
    END; SW1 = TRUE RETURN;

READ__CARD(CI.EOG.EOF.CRITEM.SW1).IF SW1 = FALSE THEN
    NEW__CARD. = READ STF;
        IF NEW__CARD__ITEM ≠ CRITEM THEN
            DO: EOG = TRUE, CRITEM = NEW__CARD__ITEM;
          SW1 = TRUE                      END;
        ELSE DO. CI: = NEW__CARD;SW1 = FALSE. END.
        IF CI = EOF__STF THEN EOF = TRUE; RETURN;
```

Figure 27. Quick fix no. 4.

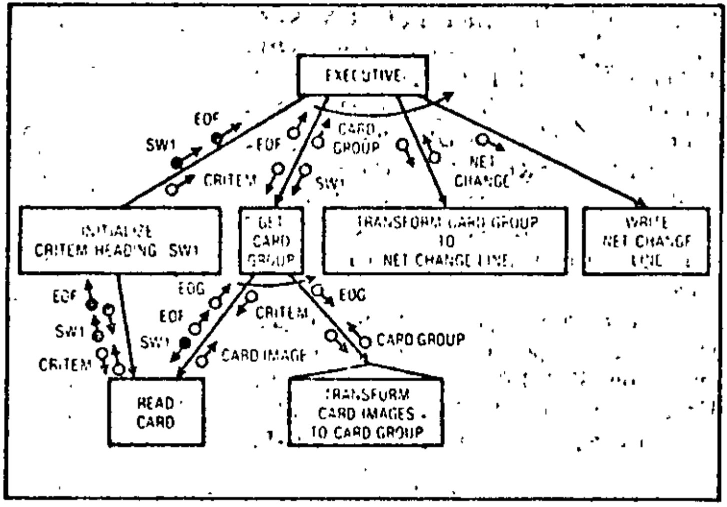


Figure 28. Data flow design with fixes.

```
EXECUTIVE  EOF. = FALSE; WRITE HEADING, SW1: = FALSE;
        CRITEM = XXXX. CI: = READ STF;
        DO WHILE (CI NOT EOF – STF);
            EOG = FALSE I. = 0.
            DO.WHILE (EOG = FALSE): I: = I + 1;
                IF SW1 = FALSE THEN NEW__CARD = READ STF;
                IF NEW__CARD__ITEM ≠ CRITEM THEN
                    DO: EOG = TRUE. CRITEM: = NEW__CARD__ITEM.
                    SW1: = TRUE:
                    END;
                ELSE DO: CI: = NEW__CARD SW1: = FALSE;
                    END.
                CG(I): = CI;
            END;
            SW1: = TRUE:
            TRANSFORM__CG__TO__NC__LINE (CG.NC);
            WRITE__NC__LINE (NC.EOF);
        END;
        STOP;
```

Figure 29. Data flow design packaged in one module.

read and write operations should be thought of as general-purpose routines callable from anywhere within the program structure. To hope to constrain them to the "ears" of a structure chart seems, at best, unwise.

Optimization. In this program, things look much better if we simply package the whole thing as one module, as shown in Figure 29.

The point, however, is that the problems Ivy had were representative of larger problems that could appear in larger programs each time the data flow design method is applied.

Data structure design

Slightly different forms of the data structure design method were developed concurrently by Michael Jackson[7] in England and J. D. Warnier[23] in France. In this discussion, Jackson's formulation and notation are used.

The basic premise is that a program views the world through its data structures and that, therefore, a correct model of the data structures can be transformed into a program that incorporates a correct model of the world. The importance of this view, stated earlier as the principle of correspondence, is emphasized by Michael Jackson's words that "a program that doesn't directly correspond to the problem environment is not poor, is not bad; but is wrong!"

When the program structure is derived from the data structure, the relationship between different levels of each resulting hierarchy tends to be a "is composed of" relationship. For example, an output report is composed of a header followed by a report body followed by a report summary. This is generally a static relationship that does not change during the execution of the program, thus forming a firm base for modeling the problem.

Since a data-structure specification usually lends itself well to being viewed as correct or incorrect, the program structure based on a data-structure specification can often be viewed as being correct or incorrect. Jackson[7] purports that two people solving the same problem should come up with program structures that are essentially the same. Thus, this method satisfies the principle of consistency to a large degree.

Design strategy. The programming process can be partitioned into the following steps:

(1) Form a system network diagram that models the problem environment.
(2) Define and verify the data-stream structures.
(3) Derive and verify the program structures.
(4) Derive and allocate the elementary operations.
(5) Write the structure text and program text.

These steps can usually be performed and verified independently, separating concerns by partitioning both the design process and the problem solution. For large problems, the objective is a network of hierarchies, each representing a simple program. Jackson's premise is that although simple programs are difficult to write, complex programs are impossible. The trick, then, is to partition complex problems into simple programs.

These simple programs, individually implemented as true hierarchical modular structures, are connected in a data-flow network. This network can be placed into a "calls" or "is called by" hierarchy by a scheduling procedure called "program inversion"[7] that is performed as a separate step after the structure of the program has been defined.

The system network diagram for a simple program is represented schematically in the upper left corner of Figure 30 and is explained further in the example. In its simplest form, it represents a network of functions that consume, transform, and produce sequential files. Below this network diagram, the data structure of each file is shown to be represented by data structure diagrams. The notation used is similar to that shown in Figure 4.

If the data structures correspond well, a program structure diagram can be drawn that encompasses both data structures. When a diagram cannot encompass both data structures, a *structure clash*[7] exists.

Since the program structure models the data structures, and since most operations are performed on data elements, one can list and allocate executable operations to each component of the program structure. These elementary operations are denoted by the small squares in the structure diagram and are shown in a list in the lower left corner of Figure 30.

The basic data structure design procedure can also be represented schematically, as shown in Figure 31. The arrows represent the flow of work and the results required in following the basic design procedure. Note that the final program structure is formed by first finding data structure correspondences and then by adding in the executable operations also derived from the data structures. The problem model is usually documented by a system network diagram, the data structures and program structure by structure diagrams, and the program text by structure text. Examples of each of these are given below in the McDonald's example.

The major problem with Jackson's data structure design methodology is that, in Jackson's words, "it is being developed from the bottom up." That is, although it is clear at this point how to apply it to small problems, the "correct" method for extending it to large system problems is still being developed.

McDonald's example. Ida is a data structure designer from way back. She went to London to study the Jackson design method. She learned French just so she could read Warnier's six paperbacks. (She says they lose a lot in translation.) With the McDonald's problem, she is certain that a data structure design is the only way to go—after all, it fits into Jackson's "stores movement"[8] problem solution format.[7]

Model step. The system network diagram for the McDonald problem was given at the bottom of Figure 11.

Data step. The second design step was to construct and verify the input and output data structures for each file shown in the system network diagram. The three basic program constructs of sequence; iteration; and selection—shown in Figure 4—apply equally well to describing the structure of a data file (see Figure 32). Note that the SORTED TRANSACTION FILE is an iteration of ITEM GROUP, which is an iteration of TRANSACTION RECORD, which—in turn—is a selection of a RECEIVED RECORD or a DISTRIBUTED RECORD. The output report is a sequence of the REPORT HEADING followed by the REPORT BODY followed by the REPORT SUMMARY. The REPORT BODY is an iteration of REPORT LINE. (Note that REPORT BODY is an iteration of REPORT LINE—not REPORT LINES. Plural names in data component boxes often indicate an error in naming.)

After the input and output data structures were diagrammed, one-to-one correspondences were shown by arrows. For example, one SORTED TRANSACTION FILE is consumed in producing one REPORT, and one ITEM GROUP is consumed in producing one REPORT LINE.

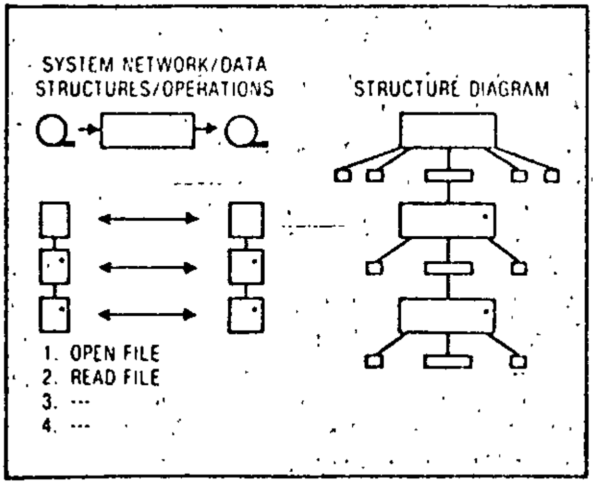


Figure 30. Data structure design method.

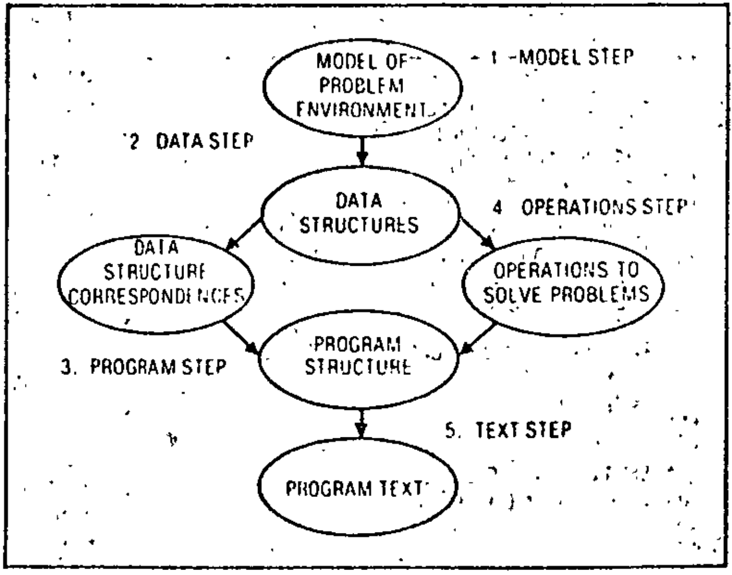


Figure 31. Basic data structure design procedure.

Program step. From these data structures, a program structure was constructed encompassing all of the parts in each data structure. Where there were one-to-one correspondences, the modules took the form of CONSUME. . . TO PRODUCE. . . .Where there were modules corresponding to only the input data structure, the form was CONSUME. . . .Where there were modules corresponding to only the output data structure, the form was PRODUCE. . . .All of these are shown in Figure 32.

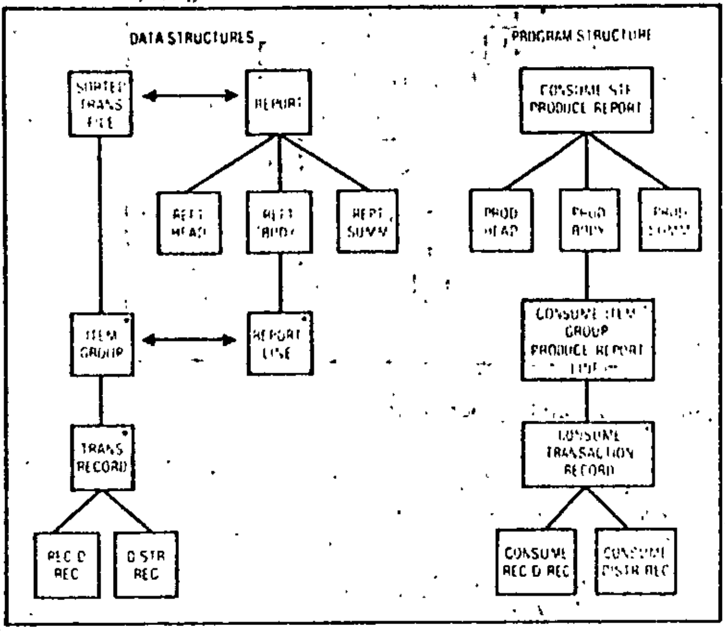


Figure 32. Data and program structures.

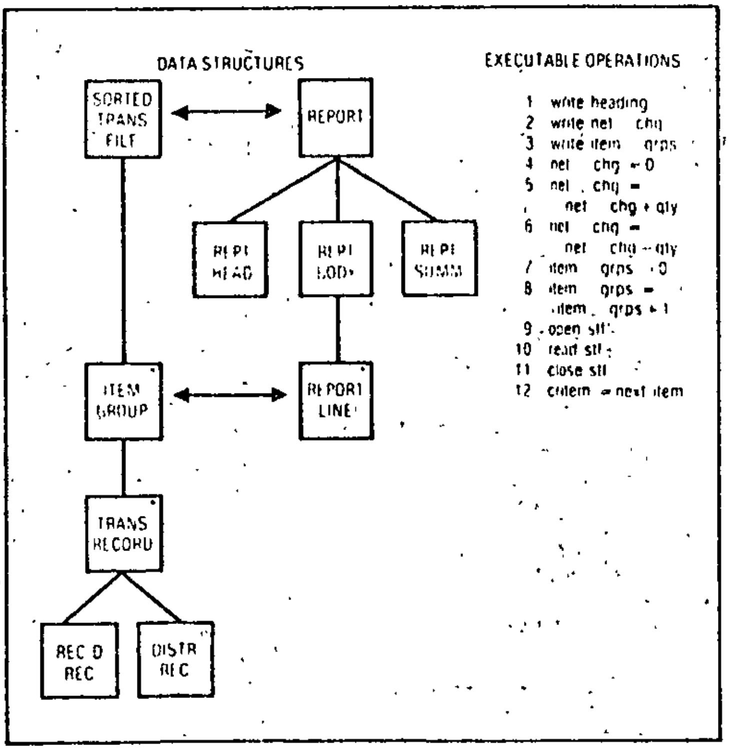


Figure 33. Listing executable operations.

In finding the producer-consumer correspondences, the question is always, "Does one instance of this result in one instance of that?" This question should be asked again during the verification procedure, which consists of showing that both the input and output data structures are subtrees of the program structure.

Operations step. The input, output and computational operations are identified by working back from the output data structure to the input data structure (see Figure 33). If necessary, the equivalent of a data flow diagram can be drawn with nodes to represent intermediate variables.

The operations must be sufficiently rich to guarantee that each output can be produced and each input can be consumed. Computational operations must also be specified that will implement the algorithms which link the two. One other type of operation is usually required for completeness. In our example, this is shown as operation 12. This operation provides the key that will be used in determining item group boundaries, and it is usually called a "structure-derived operation."

While it is desirable to have a complete list of operations from the start, it is not required, since missing operations will become apparent later in the process.

The last part of the operations step involves allocating all of these executable operations to the program structure (see Figure 34). In each case, the questions to ask are "How often should this operation be executed?" and "Should this operation occur before or after. . .?" For this example, the answer to the first question could be once per report, once per sorted transaction file, once per heading, once per summary, once per item group, once per report line, once per transaction record, once per item received record, or once per item distributed record. Clearly, you open the sorted transaction file, or STF, before you read it, you close the STF before you stop, etc. Using the read-ahead rule,[7] you read one card ahead and then read to replace (see operation 10).

At the end of this step, one should verify that all outputs are produced, all intermediate results are produced, and all inputs are consumed.

Text step. Although the structure of the program is now secure, Jackson recommends one final step before coding. That fifth step is to translate the structure diagram into structure text, as shown in Figure 35. This is done using the three basic program constructs shown in Figure 4.

Structure text is straightforward and easy to understand as long as your program labels are short and descriptive. With long program labels, it can become a mess. In Figure 35, the letters C and P are sometimes used as abbreviations for CONSUME and PRODUCE.

Ida's final program was produced by translating the structure text of Figure 35 into the target programming language of Figure 36. The major disadvantage of this method is that the number of distinct steps and procedures involved imply that both the data structure diagram and the structure text should be kept as permanent parts of the documentation. This seems unlikely unless an automated structure chart drawer is available that will store the structure chart with the program in a convenient form so that it can be updated when the program is changed. At this point, available machine aids are far from adequate. Another disadvantage is that for certain classes of problems, the data structure diagrams and the resulting program structures can get unduly cumbersome.

A programming calculus

While a "proof of correctness" is disappointingly difficult to develop after a program has been written, the constructive proof-of-correctness discipline taught by Dijkstra[20] and Gries[24] is relatively encouraging. Dijkstra's design discipline can be methodically applied to obtain a modest-sized "elegant" program with a "deep logical beauty." Using this method, the program and the proof are constructed hand-in-hand.

Design strategy. The initial design task consists of formally specifying the required result as an assertion stated in the predicate calculus. Given this desired postcondition, one must derive and verify the appropriate preconditions while working back through the program being constructed. The program and even individual statements play a dual role in that they must be viewed in both an operational way and as predicate transformers. The method is a top-down method to the extent that both the resulting program and the predicates can be formed in stages by a sequence of stepwise refinements.

Definitions. The programming language is used as a set of statements that perform predicate transformations of the form[20,24,25]

$$\{Q\}S\{R\} \qquad (1)$$

In this expression, Q represents a precondition that is true before execution of statement S. R represents a postcondition that is true after the execution of statement S. In simple terms, the formula means that the execution of statement S beginning in a state satisfying Q will terminate in a state satisfying R, but Q need not describe the largest set of such states. When Q does describe the largest set of such states, it is called the weakest (least restrictive) precondition.

When Q is the weakest precondition, this is expressed in the form

$$Q = wp(S, R) \qquad (2)$$

When S is an *assignment* statement, it takes the form

$$x := e \qquad (3)$$

where e is an expression. Its definition as a predicate transformer is given by

$$wp("x := e", R) \equiv R_e^x \qquad (4)$$

The term R_e^x is used to denote the textual substitution of expression e for each free occurrence of x within expression R.

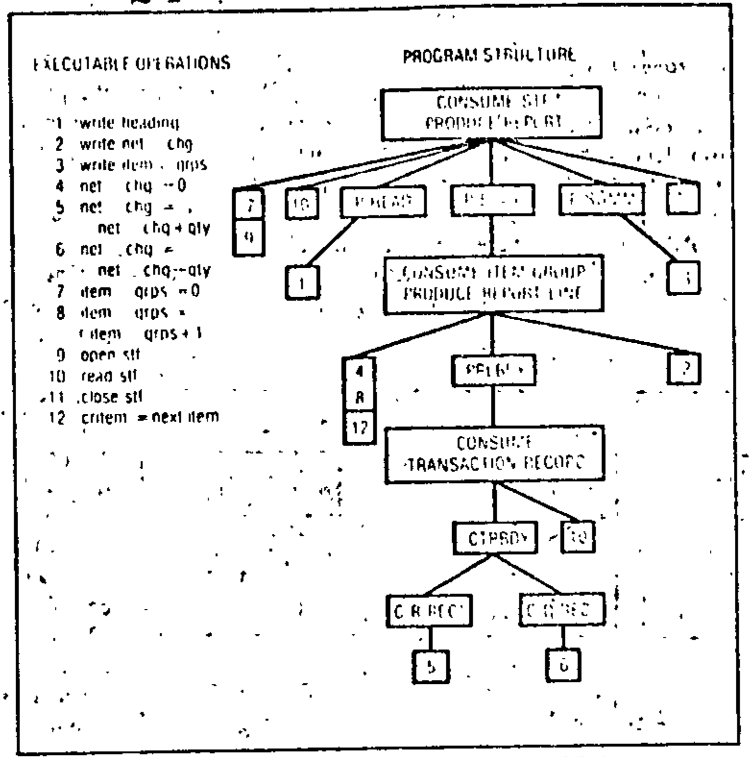


Figure 34. Allocating executable operations.

When s is a *Selection* statement, it takes the form

$$\begin{aligned} \text{IF} \equiv \textbf{if}\ & B_1 \rightarrow S_1 \\ [\!]\ & B_2 \rightarrow S_2 \\ & \cdots \qquad\qquad (5) \\ [\!]\ & B_n \rightarrow S_n \\ \textbf{fi}\ & \end{aligned}$$

```
P seq
    item__grps. = 0.
    open stf; read stf;
    write heading.
    P-BODY itr until (eof-stf)
      C-ITEM GRP-P-REPT-LINE seq
        net__chg: = 0
        item__grps: = item__grps + 1.
        critem. = next item.
        PRLBDY itr while (next item = critem)
          C-TRANS-REC seq
            CTRBDY sel (code = R)
              net__chg: = net__chg + qty.
            CTRBDY alt (code = D)
              net__chg: = net__chg − qty.
            CTRBDY end
            read stf
          C-TRANS-REC end
        PRLBDY end
        write net__chg.
      C-ITEM GRP-P-REPT-LINE end
    P BODY end
    write item__grps. close stf.
P end
```

Figure 35. Structure text.

This is valid where $n > 0$, the B_i are boolean expressions called guards, and the S_i are statements. The vertical bar "$[\!]$" serves to separate otherwise unordered alternatives. To execute IF, at least one of the guards B_i must be true. To execute it, choose one guard that is true (nondeterministically) and execute the corresponding statement. The expression $B_1 - S_1$ may be read as "When guard B_1 is true, statement S_1 may be executed." The definition of IF as a predicate transformer is given by

$$wp(\mathrm{IF}, R) \equiv \mathrm{BB} \text{ and } (\mathrm{A}\, i : 1 \leq i \leq n : B_i => wp(S_i, R)) \quad (6)$$

where $\mathrm{BB} \equiv B_1$ or B_2 or . . . or B_n. Thus, we must prove that executing any S_i when B_i and Q are true establishes R. In practice, given R and S_i, we derive B_i using the weakest precondition.

The *iteration* statement takes the form

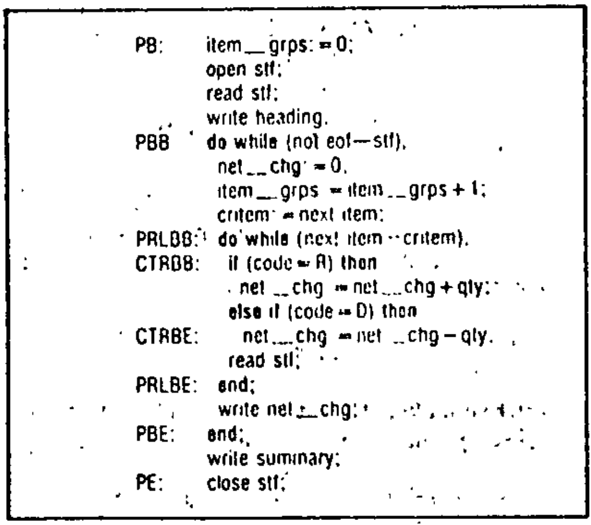
$$\begin{array}{lll} \mathrm{DO} & \equiv & \textbf{do } B_1 - S_1 \\ & & [\!] \quad B_2 - S_2 \\ & & \quad . \; . \; . \\ & & [\!] \quad B_n - S_n \\ & & \textbf{od} \end{array} \quad (7)$$

Iteration continues as long as at least one of the guards $B_1, \ldots, B_n$ is true. From the set of statements with true guards, one is selected (nondeterministically) for execution. The definition of DO as a predicate transformer is given by

$$wp(\mathrm{DO}, R) \equiv (\mathrm{E}\, k : k \geq 0 : \mathrm{H}_k(R)) \quad (8)$$

where $\mathrm{H}_k(R)$ for $k \geq 0$ is the weakest precondition such that execution of DO will terminate after, at most, k "iterations" and in a state satisfying R. In developing a program that contains a loop structure, this is found by completing the following steps:

1. Write a formal specification of the desired result R.
2. Determine an invariant P.
3. Derive a loop body such that:
   a. $Q => P$ (i.e., P is initially true).
   b. $\mathrm{A}\, i : \{ P \text{ and } B_i \} S_i \{ P \}$
   (i.e., P remains invariantly true).
   c. $(P \text{ and not BB}) => R$
   (i.e., upon termination R is true).
   d. Show the loop terminates:
      (1) $(P \text{ and BB}) => t > 0$
      (i.e., before termination, t is positive).
      (2) $\mathrm{A}\, i : \{ P \text{ and } B_i \} \tau := t; S_i \{ t \leq \tau - 1 \}$
      (i.e., t decreases with each iteration).



```
        PB:     item__grps: = 0;
                open stf;
                read stf;
                write heading.
        PBB     do while (not eof—stf),
                  net__chg: = 0.
                  item__grps = item__grps + 1;
                  critem: = next item;
        PRLBB:    do while (next item = critem),
        CTRBB:     if (code = R) then
                    net__chg = net__chg + qty;
                   else if (code = D) then
        CTRBE:       net__chg = net__chg − qty.
                   read stf;
        PRLBE:    end;
                  write net__chg;
        PBE:    end;
                write summary;
        PE:     close stf;
```

Figure 36. Final data structure design program.

In step 3d, t is a termination function chosen to be positive during the execution of the iteration and zero upon completion; τ is a fresh integer variable.

McDonald's example. The problem is initially simplified by eliminating the R/D field of each transaction and considering the quantity field $q(j)$ as positive for items received and negative for items dispersed. The R/D field is reintroduced later when the final pseudocode is written so that the resulting program can be readily compared with previous designs.

Problem specification. Note that the STF is an iteration of item group, which is an iteration of transaction, which—in turn—is a sequence of the item field $f(j)$ and the quantity field $q(j)$ for $1 \leq j < m$ and $f(m) = EOF$.

Since the transaction file has been sorted, an item group $g(j,k)$ can be defined as

$$g(j,k) \equiv f(j) \neq f(j-1) \text{ and } f(j) = f(k) \text{ and } f(k) \neq f(k+1) \quad (9)$$

Similarly, a partial item group $\hat{g}(j,k)$ can be defined as

$$\hat{g}(j,k) \equiv f(j) \neq f(j-1) \text{ and } f(j) = f(k) \text{ and } f(k) = f(k+1) \quad (10)$$

The number of complete groups over the range 1 through k can be defined as

$$n(1,k) \equiv (\mathrm{N}\, p : 1 \leq p \leq k : f(p) \neq f(p+1)) \quad (11)$$

where $\mathrm{N}\, p$ is the number of distinct values of p over the range $1 \leq p \leq k$ for which $f(p) \neq f(p+1)$.[5]

Formal specification of R. Step 1 in the design procedure is to develop, in a top-down fashion, a formal specification of the desired result R. A first attempt could look something like this:

R : The total number of item groups i is calculated. The net change in inventory c is calculated for each item group.

More formally:

$$R : (\mathrm{A}\, j,k : 1 \leq j \leq k < m \text{ and } g(j,k) \\ : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \quad (12)$$

This whole expression may be read "If R is true, the following holds: For all j and k such that $1 \leq j \leq k < m$ and the transactions between j and k inclusive form a group, it is true that i is set equal to the number of complete groups over the interval 1 through k and $c(i)$ is formed as the sum of the quantity fields within each complete group." This notation is consistent with that used by Dijkstra and Gries.[25] Note that the result assertion specifies *what* is to be derived—not *how* it is to be derived.

When we examine (12), the requirement for counting the number of complete groups and the summation required for computing $c(i)$ suggest that at least one loop structure is required in the program. Thus, our next step is to determine an invariant P.

Determine an invariant P. When step 2 is performed, the result assertion (12) is weakened to form an invariant P. In this example, R may be weakened by introducing a variable $\hat{m}$, where $1 \leq \hat{m} \leq m$, such that

$$\begin{aligned} P:&(1 \leq \hat{m} \leq m) \\ &\text{and } (\text{A } j,k : 1 \leq j \leq k < \hat{m} \text{ and } g(j,k) \\ &\qquad : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \\ &\text{and } (\text{A } j,k : 1 \leq j \leq k < \hat{m} \text{ and } \hat{g}(j,k) \text{ and } k = \hat{m} - 1 \\ &\qquad : \hat{c} = \sum_{r=j}^{k} q(r)) \end{aligned} \tag{13}$$

This invariant states that "If P is true, the following holds: For all the complete item groups, i is set to the group number and $c(i)$ is set to the sum of the q fields within each group. For the last partial item group, $\hat{c}$ is set to the sum of the q fields."

Derive the program. Step 3d(1) requires that a termination function t be chosen that is greater than zero while any of the guards B_i are still true and that decreases with every execution of S_i. For our example, a termination function that can readily be made to satisfy these conditions is

$$t = m - \hat{m} \tag{14}$$

This function will equal zero on termination if the guard B_1 is set to terminate the iteration when $\hat{m} = m$.

With this in mind, the basic program form becomes

```
"define Q to establish P";
{P}
do $\hat{m} \neq m$ —
   "decrease t while maintaining P"                (15)
od
{R}
```

A statement that would decrement t is

$$\hat{m} := \hat{m} + 1 \tag{16}$$

As noted in step 3a, we must next show that P is initially true. In other words, the initial state $\{Q\}$ must be so defined that Q implies P. That is

$$Q => P \tag{17}$$

Note that if the program starts with the statement

$$\hat{m}, i, \hat{c} := 1,0,0 \tag{18}$$

then P holds.

To satisfy step 3b, we must show that P remains true after statement (16) is executed. One way of meeting this requirement is to introduce further guards B_i by embedding an IF selection statement within the loop construct. The IF statement format is shown in (5). The program is now in the form

```
$\hat{m}, i, \hat{c}$ := 1,0,0;
{P}
do $\hat{m} \neq m$ —
   if
         "statements that reestablish P"           (19)
   fi;
   $\hat{m}$ := $\hat{m}$ + 1
   {P}
od
{R}
```

The guards of the IF statement must be chosen such that P remains true after statement (16) is executed. In general, guards B_i must be chosen so that

$$P \text{ and } B_i => wp(S_i, P) \tag{20}$$

In our example, we must find

$$\begin{aligned} &wp(\text{"}\hat{m} := \hat{m} + 1\text{"}, P) \\ &\quad = (1 \leq \hat{m} + 1 \leq m) \\ &\qquad \text{and } (\text{A } j,k : 1 \leq j < k < \hat{m} + 1 \text{ and } g(j,k) \\ &\qquad\qquad : i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)) \\ &\qquad \text{and } (\text{A } j,k : 1 \leq j \leq k < \hat{m} + 1 \text{ and } \hat{g}(j,k) \\ &\qquad\qquad \text{and } k = \hat{m} \\ &\qquad\qquad : \hat{c} = \sum_{r=j}^{k} q(r)) \end{aligned} \tag{21}$$

Consequently, we must choose guards for the IF that ensure that all three of the terms of (21) are satisfied when (16) is executed. The first term

$$1 \leq \hat{m} + 1 \leq m \tag{22}$$

can be established directly from P and the iteration guard ($\hat{m} \neq m$). The second term of (21)

$$\text{A } j,k : 1 \leq j \leq k < \hat{m} + 1 \text{ and } g(j,k) \tag{23}$$

27

$$: i = n(1,k) \text{ and } c(i) = \sum_{r=j}^{k} q(r)$$

can be established if P is true and if

$$f(\hat{m}) \neq f(\hat{m}+1) \rightarrow i := i+1; c(i) := \hat{c} + q(\hat{m}); \hat{c} := 0 \tag{24}$$

where initially

$$\hat{c} = \sum_{r=j}^{\hat{m}-1} q(r) \tag{25}$$

It can also be shown that the third term of (21)

$$\mathbf{A}\ j,k : 1 \leq j \leq k < \hat{m}+1 \text{ and } \hat{g}(j,k) \text{ and } k = \hat{m} \tag{26}$$

$$: \hat{c} = \sum_{r=j}^{k} q(r)$$

is true if P is true and if

$$f(\hat{m}) = f(\hat{m}+1) \rightarrow \hat{c} := \hat{c} + q(\hat{m}) \tag{27}$$

With these observations, our final program becomes

```
$\hat{m}, i, \hat{c} := 1,0,0;$
$\{P\}$
do $\hat{m} \neq m \rightarrow$
    if $f(\hat{m}) = f(\hat{m}+1) \rightarrow \hat{c} := \hat{c} + q(\hat{m})$                    (28)
    $[] f(\hat{m}) \neq f(\hat{m}+1) \rightarrow i := i+1; c(i) := \hat{c} + q(\hat{m}); \hat{c} := 0$
    fi;
    $\hat{m} := \hat{m}+1 \{P\}$
od
$\{R\}$
```

Show the loop terminates. During the design procedure, we have verified the truth of the first three conditions stated earlier in step 3 regarding the invariant P. Step 3d can be satisfied by showing that t remains positive while at least one guard is true and that each iteration decreases t. These final termination conditions should be verified by the reader.

Pseudocode. A pseudocode version of the final program is shown in Figure 37. Note that the meanings of code R, code D, and EOF have been restored. In this figure, i becomes *item_grps*, $\hat{c}$ becomes *net_chg*, $\hat{m}$ corresponds to *curr_rec*, and $f(\hat{m})$ corresponds to *curr_item*.

Conclusion

Comparison. While there is something to be learned from examining the pseudocode resulting from the four solutions to the McDonald's problem, more significant points can be made by comparing the differences in program structure.

Functional decomposition solution. A skeleton structure diagram for the functional decomposition solution is shown in Figure 38. Note that the basic operation was assumed to be PROCESS CARD and that this design treats the first card as different from the subsequent cards quite early. This results in a structure that favors the addition of changes that can be keyed to the beginning of a group. Changes that must be keyed to the end of a group are added with more difficulty.

Data flow design solution. The data flow design solution had the basic structure shown in Figure 39. Note that in this case there is a difference between a lead card and the last card in a group. That is the last card in a group gets treated in a special way—not the first card. For later modification, this means that operations added to the end of a group will be favored while operations added to the beginning of a group will not fit in as naturally.

Even through the concept of group is well localized on the diagram, the data flow nature of the processing forces one to do some of the group operations at the PROCESS CARD level. In this case, an end-of-group control signal must be inserted at the PROCESS CARD level so that other end-of-group processing functions can be activated within the PROCESS GROUP function.

The resulting program has sequential cohesion. Both data and control are passed up the structure diagram. Generating the end-of-group indication still causes a few minor problems, but in this particular example keying on the last card in a group turns out to be a better choice than keying on the first card in a group.

Data structure design solution. The data structure design solution is shown in Figure 40. In this solution, all of the cards are processed by the same PROCESS CARD function. The keying of groups is done automatically

```
item__grps := 0.
net__chg: = 0;
open stf;
curr__rec: = read stf;
write heading
do curr__rec ≠ EOF
   next__rec = read stf;
   if curr__item = next__item →
      if code = R → net__chg = net__chg + qty.
      ▮ code = D → net__chg: = net__chg – qty:
      fi
   ▮ curr__item ≠ next__item →
      item__grps: = item__grps + 1;
      if code = R → net__chg. = net__chg + qty:
      ▮ code = D → net__chg = net__chg – qty.
      fi
      write net__chg:
      net__chg = 0:
   fi
   curr__rec = next__rec;
od
write item__grps.
close stf;
```

Figure 37. Programming calculus McDonald's solution.

within the control structure through use of the read-ahead rule. Thus, no cards are treated with special functions, and everything looks clean. This solution can handle further additions that require special action at the beginning or at the end of a group with relative ease.

Programming calculus solution. The programming calculus solution structure diagram is shown in Figure 41. Note that this solution treats the last card as special, favoring extensions that occur at the end of a group. Since most of the McDonald's problem group functions are end-of-group functions, the resulting solution looks quite clean. In the long run, this solution could cause problems when new beginning-of-group operations are added.

Of these four solutions, the one that seems to model the problem best is the data structure design solution shown in Figure 40. A proper choice of loop invariants using the predicate calculus may have led to the same structure, but the choice that was made didn't. Likewise, a proper application of functional decomposition also could have led to the same structure. Even the data flow design method does not completely rule out this preferred solution.

It might be argued that all four design methodologies were capable of yielding any of the other solutions. My view is that it would take a designer with unusual insight to reliably proceed toward the preferred solution using any of these techniques. I do feel, however, that for the class of problems represented by the McDonald's example, people are more likely to derive the preferred solution

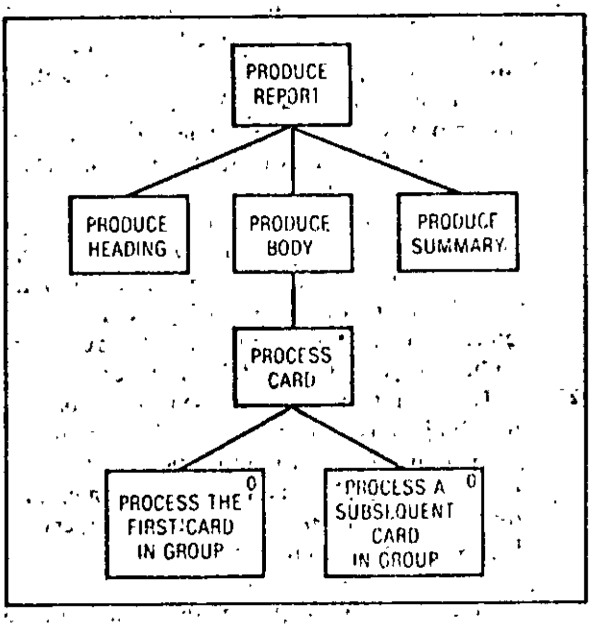


Figure 38. Functional decomposition structure diagram.

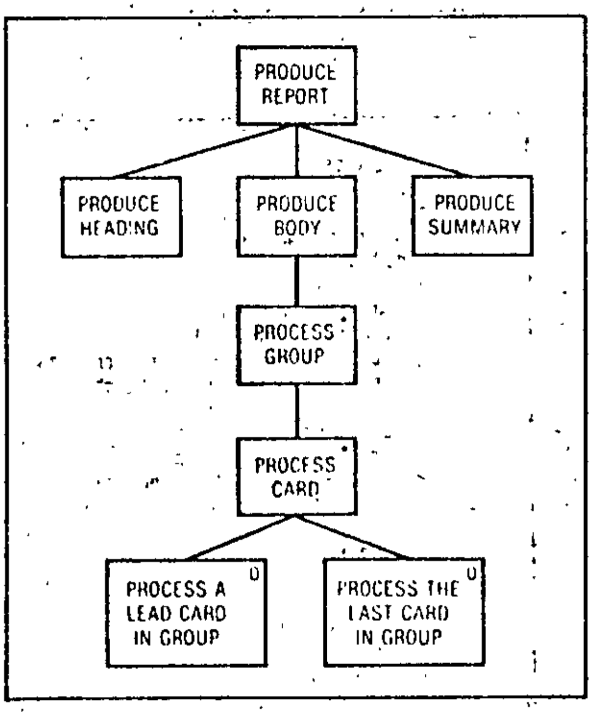


Figure 39. Data flow structure diagram.

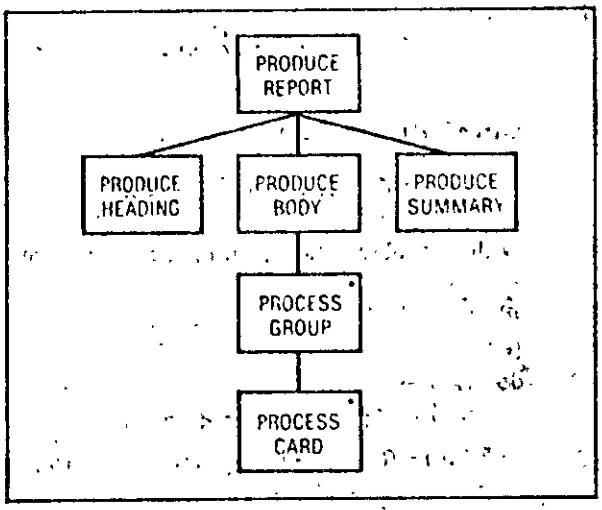


Figure 40. Data structure structure diagram.

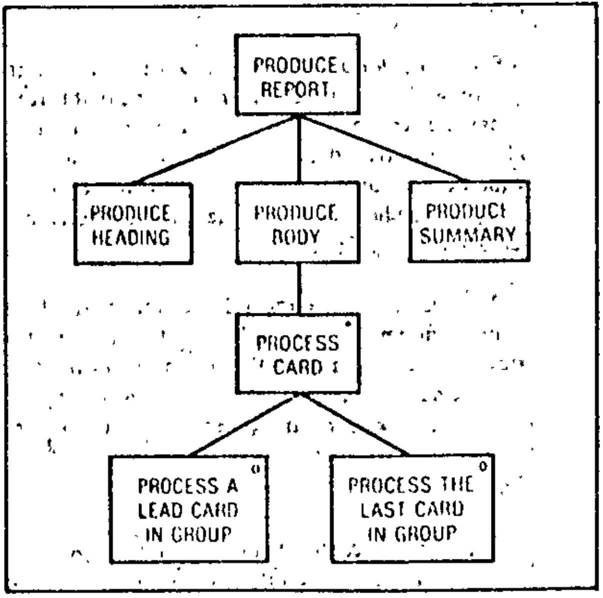


Figure 41. Programming calculus structure diagram.

by using data structure design than by using any of the other methods. For a different class of problems, of course, the results could be different.

2) **Critique.** Since functional decomposition has been around for more than a decade, there have been many well-documented success stories and even a few well-documented failures.

One success story was summarized by its designers, using the diagrams in Figure 42. Perceived project visibility was dramatically improved by the application of functional decomposition together with other techniques. They also felt that project staffing could be reduced over that normally required. In this particular project, there was a lot of personnel turnover but this was taken in stride, partly owing to the beneficial effects of the new techniques.

It seems that functional decomposition can lead to a "good" hierarchical program structure if carefully applied. If not used carefully, however, it can lead toward logical cohesion and, occasionally, towards telescoping—that is, toward defining smaller and smaller modules that are not independent but have strong coupling with each other. Applying functional decomposition to obtain mathematical functions (e.g., square root) is relatively straightforward.

For any given problem, the number of potential decompositions can be large. This makes applying the technique much more of an art than a science. I know that Dijkstra can do beautiful functional decompositions.

In Johnson's[11] words, functional decomposition seems to be a triumph of individual intellect over lack of an orderly strategy. The question "Decomposition with respect to what?" is always a point to ponder. The measures of "goodness" are difficult to apply consistently. Finally, this method requires that the intellectual tasks of problem modeling and program construction be addressed simultaneously. Ideally, these two tasks would be separated.

2) Because the concept of data flow design came later than pure functional decomposition, it has not been used as extensively. Apparently, several projects within IBM have used the "composite design" version of data flow design with varying degrees of success. I have personally seen a number of success stories that praise data flow design.

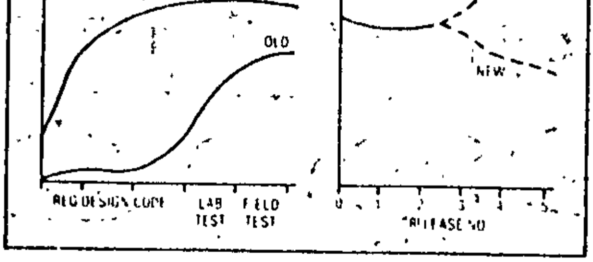


Figure 42. Experience with functional decomposition as used with hierarchical structure, stepwise refinement, high-level language, teams, walkthroughs, cause/effect charts, etc.

The concepts of coupling and cohesion, which accompanied the introduction of the data flow design method,[8] have been thought-provoking and, on some occasions, revealing. They have given people a language to express previously unverbalized thoughts.

The data flow design method can be used to produce a hierarchical program structure with all of its intrinsic advantages. The tendency is strongly toward modules with sequential cohesion at the system level, although anything can happen within the central transform. The data flow chart, which forms the basis for decomposing with respect to data flow, is a useful contribution and may be the best approach currently available at the system design level. It's not clear that the later step of putting things into a "calls" hierarchy using "transform centered design"[8] is as useful. It seems to produce a structure with a lot of data passing and adds artificial "afferent" (input) and "efferent" (output) ears to the structure chart, while reverting back to standard functional decomposition for the "central transform" which is the heart of the problem. It isn't clear that anything is gained over simply using functional decomposition from the start.

In summary, the concepts of cohesion and coupling represent a real step forward from straight functional decomposition. A qualitative measure of goodness is not as good as a quantitative measure, but it is a start. Furthermore, the data flow chart separates the modeling of the problem from detailing the structure of the program. The hard part becomes deriving the correct data flow chart (or "bubble chart"), which is still an art for large systems.

3) Neither the Jackson nor the Warnier methodologies for data structure design were widely used in this country until quite recently. They have, however, been used for a number of years in Europe. Jackson's method seems closer to a true methodology than the other design methodologies currently available. It is repeatable, teachable, and reliable in many applications. It usually results in a program structure that faithfully models the problem.

The data structure design method results in a hierarchical program structure, if the data structure is hierarchical. It produces multiple, independent hierarchies, if they are present in the problem environment. It's difficult to determine the level of cohesion of the modules within the resulting program structure. Sometimes it tends to be functional, in other cases communicational. By modeling the data structures and therefore the problem environment first, the problem modeling task is done before the program construction task.

While there is still no clear methodology for large systems and deriving the "correct" data structures can be difficult, it still seems that this method is a big step forward. Being able to ask whether a structure is right or wrong is somehow much more satisfying than trying to decide if it is good, better, or best.

Both Jackson's and Warnier's data structure design methods were first applied in business data processing. At this point, they have also been applied to a number of online problems, although they are still unproven for large real-time applications.

4) The number of people who use the programming calculus method regularly and proficiently is extremely

small. The primary disadvantage is that a relatively high degree of logical and mathematical maturity is required to produce even "simple" programs. The mathematical proofs involved are usually several times longer than the program derived.

A second and perhaps less important disadvantage is that this method admits the existence of multiple solutions to the same problem. Different choices of an invariant assertion can lead to different program structures. The resulting programs do not necessarily portray accurate and consistent models of the problem's environment or its solution. That is, a "correct" program may still have the "wrong" structure.

In spite of these problems, Dijkstra's programming calculus design discipline is an encouraging step forward on the road to developing correct programs. It is a method that you should be aware of, for it holds promise for the future.

Summary. As shown in Figure 43, many claims have been made about the different strategies for designing software. For functional decomposition, the proponents have largely said, "D is good design, believe me." For data flow design methods, people have said, "Program C is better than program D. Let me tell you why." For data structure design methods, the claim is that "B is right; C and D are wrong. A program that works isn't necessarily right." In the programming calculus, the contention is that "Program A is probably correct. B, C, and D are unproven."

In my view, there is still much room for innovation in the area of program design methodologies. The current state of the art was represented schematically by Johnson[11] in the form of Figure 44. Functional decomposition has been described as the ideal methodology for people who already know the answer. The other three methodologies seem less reliant on knowing the answer before you start.

All of the methodologies rely on some magic. For functional decomposition, the magic gets applied very close to the end product. In the other three methodologies, at least some of the magic gets applied at the problem-model level. Clearly, we need to get a much better handle on the magic part of all four of the methodologies.

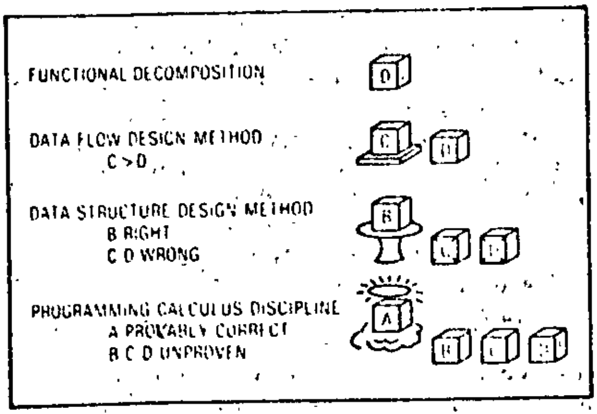


Figure 43. Summary of program design methodology claims.

Prognosis. Many different design methodologies are available. Although the first three methodologies discussed in this article have been used extensively and to a large measure successfully, there is still much to be done.

Desired results. The results I would like to see from the work yet to come are

- a complete methodology for partitioning "big" problems,
- better documentation for both the shortcomings and attributes of existing methodologies,
- guidelines for combining methods when appropriate,
- a generally accepted metric for quantifying program complexity (or entropy),
- help for the four out of five programmers who are maintaining and enhancing old programs, and
- more published examples of real-time applications.

Hopes for the future. Some possible requirements on a "complete" methodology could include the following:

- It must include a rational procedure for partitioning and modeling the problem.
- It should result in consistent designs when applied by different people.
- It must systematically scale upward to large problems while interacting consistently with a model of the real world.
- It must partition the design process as well as the problem solution.
- The correctness of individual design steps must guarantee the correctness of the final combination.
- It should minimize the innovation required during the design process. The innovation should occur in the algorithm specification phase.

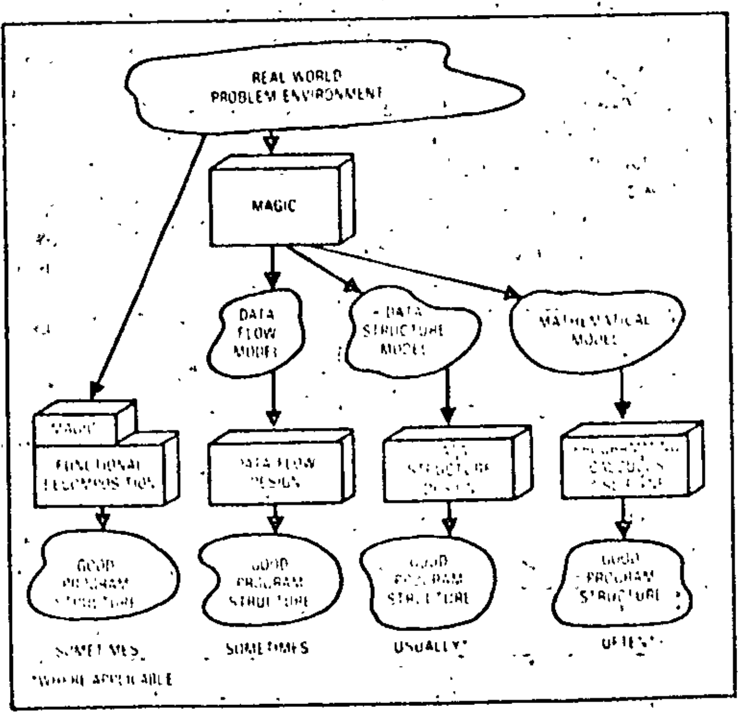


Figure 44. Current state of the art.

An interim procedure. Until we know the "right" method for designing the structure of a large program, I would propose the following interim procedure:

- Define the high-level language and operating system macros bottom-up, using the principle of abstraction to hide the peculiarities of the hardware and to create a desirable virtual machine environment.
- Map the system flow diagram into your virtual machine environment.
- Work through the inversion[7] process or construct a data flow model of the "big" problem.
- Construct data structure diagrams that correspond to each data flow path.
- Cluster and combine bubbles that can be treated by one of Jackson's "simple" programs.
- Use the Jackson data structure design method to combine simple programs and reduce the number of intermediate files.
- Implement each cluster as concurrent, asynchronous processes if you are operating under a suitable programming environment—for example, Simula or Unix.[26]*
- If a decent operating system is not available, construct some scheduling kludge.

While these interim suggestions do not fit together well enough to call them a method, they may form a reasonable approach to follow until a true methodology for large problems is found. ■

*Unix is a trademark of Bell Laboratories.

Acknowledgments

Much of the material presented here was developed and discussed over the last several years with R. D. Gordon and J. W. Johnson. The functional decomposition and data structure design solutions to the McDonald's problem are extensions of work described by Michael Jackson. The programming calculus solution to the McDonald's problem was developed after discussions with F. Schneider, G. Levin, and R. D. Gordon. Their inputs and support are gratefully acknowledged.

Several excellent suggestions from R. R. Conners, B. Dwyer, M. R. Gornick, R. C. Hansen, W. R. Nehrlich, and D. Sharma were incorporated in the final draft of this article.

References

1. J. N. Buxton, "Software Engineering," *Programming Methodology*, ed. D. Gries, Springer-Verlag, New York, 1978, pp. 23-28.
2. B. W. Boehm, "Software Engineering," *IEEE Trans. Computers*, Dec. 1976, Vol. C-25, No. 12.
3. D. L. Parnas, "Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs," *Programming Methodology*, Lecture Notes in Computer Science, No. 23, Springer-Verlag, New York, 1975, pp. 225-235.
4. V. A. Vyssotsky, "Software Engineering," keynote speech delivered at COMPSAC 79, Nov. 6-8, 1979, Chicago, Ill.
5. G. D. Bergland and R. D. Gordon, *Tutorial: Software Design Strategies*, IEEE Computer Society Press, Silver Spring, Md., 1979, pp. 1-14.
6. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
7. M. A. Jackson, *Principles of Program Design*, Academic Press, New York, 1975.
8. E. Yourdon and L. L. Constantine, *Structured Design*, Yourdon Press, New York, 1975.
9. G. J. Myers, *Reliable Software Through Composite Design*, Petrocelli/Charter, New York, 1975.
10. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *SIGPLAN Notices*, Vol. 9, No. 4, Apr. 1974, pp. 50-59.
11. J. W. Johnson, "Software Design Techniques," *Proc. Nat'l Electronics Conf.*, Chicago, Ill., Oct. 12, 1977.
12. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Pub. Co. Reading, Mass., 1975.
13. E. W. Dijkstra, "The Humble Programmer," *Comm. ACM*, Vol. 15, No. 10, Oct. 1972, pp. 859-866.
14. *Implications of Using Modular Programming*, Central Computer Agency, Her Majesty's Stationery Office, London, 1973.
15. C. Alexander, *Notes on Synthesis of Form*, Harvard University Press, Cambridge, Mass., 1964.
16. L. A. Belady and M. M. Lehman, "Characteristics of Large Systems," *Proc. Research Directions in Software Technology*, Brown University, Oct. 1977.
17. F. T. Baker, "Structured Programming in a Production Programming Environment," *IEEE Trans. Software Eng.*, June 1975, Vol. SE-1, No. 2, pp. 241-252.

18. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, Dec., 1972, pp. 1053-1058.

19. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming, Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.

20. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

21. N. Wirth, *Systematic Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

22. B. H. Liskov, "A Design Methodology for Reliable Software Systems," *AFIPS Conf. Proc.*, Vol. 41, FJCC 72, 1972, pp. 191-199.

23. J. D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold Co., New York, 1974.

24. D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," *IEEE Trans. Software Eng.*, Vol. SE-2, No. 4, Dec. 1976, pp. 238-244.

25. E. W. Dijkstra and D. Gries, "Introduction to Programming Methodology," Ninth Institute in Computer Science, University of California, Santa Cruz, Aug. 1979.

26. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical J.*, Vol. 57, No. 6, Part 2, July-Aug. 1978, pp. 1905-1929.

Glenn D. Bergland is head of Bell Telephone Laboratories' Digital Systems Research Department in Murray Hill, New Jersey. In 1966, when he joined Bell Telephone Laboratories in Whippany, New Jersey, he conducted research in highly parallel computer architectures. In 1972, he became head of the Advanced Switching Architecture Department in Naperville, Illinois. Later, he became head of the Software Systems Department, which was involved in feature development for the No. 1 electronic switching system. Currently, his major research areas are software design methodologies, digital telecommunications services, personal computing, and nonstop computer systems.

Bergland received the BS, MS, and PhD in electrical engineering from Iowa State University in 1962, 1964, and 1966. He is a member of the IEEE and ACM.

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

CONSTRUCCION DE PROGRAMAS
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

viii. CONSTRUCCION DE PROGRAMAS
(DOCUMENTACION COMPLEMENTARIA)

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

codificación (4.2.3).

Desde el punto de vista de la programación estructurada, la mejor documentación de un programa la constituye la claridad de su estructura; además, la única documentación confiable de un programa es el programa mismo, pues sólo leyendo el código puede el programador dar por hecho lo que hace el programa. De allí el énfasis en la legibilidad del código, base indiscutible de la programación estructurada.

3.4 Técnicas De Desarrollo

El objetivo de las técnicas de desarrollo es proporcionar una guía para la codificación, integración y verificación de los módulos de un programa.

En esta sección se presentan las técnicas empleadas con mayor frecuencia en el desarrollo de programas de computadora.

3.4.1 Tecnicas De Codificación -

Las técnicas de codificación tienen como objetivo facilitar la traducción del diseño detallado de los módulos que componen un programa a un lenguaje de programación específico, manteniendo la estructura jerárquica definida en la fase de diseño.

La descripción detallada de los módulos sirve como base para la codificación. Cualquier módulo, aún el más complicado, ha sido detallado utilizando las reglas básicas de programación estructurada: los módulos tienen una sola entrada y una sola salida y puedan ser facilmente traducidos, revisados y probados en forma independiente.

El detalle de los módulos se traduce en instrucciones de máquina (computadora), dependiendo de las características del propio lenguaje de programación. Mientras algunos lenguajes incluyen las estructuras de control básicas entre sus instrucciones normales(ALGOL, PASCAL, PL/1, FORTRAN 77), en los lenguajes más populares (COBOL, BASIC, FORTRAN 66), estas facilidades no solo no existen, sino que además presentan algunas dificultades para su implementación.

La situación ideal se presenta cuando la instalación de cómputo cuenta con un lenguaje de programación estructurado, con lo cual la función de traducir el detalle de los modulos se convierte en una tarea trivial como se muestra a continuación:

```
ESTRUCTURA                               IMPLEMENTACION FORTRAN 77

Mientras (bandera='S')                   WHILE(BANDERA='S')
 Llama actualiza                          CALL ACTUALIZA
 Limpiar pantalla                         REWIND 3
 Escribir (pantalla) 'cambios (S,N)?'     WRITE(3,5000)
'CAMBIOS(S,N)?'
  Leer (pantalla) bandera                  READ(3,5100) BANDERA
Fin (mientras)                           END WHILE
```

Sin embargo, esto no sucede comunmente con los lenguajes utilizados en la mayoría de las instalaciones de cómputo. En el caso del lenguaje de programación COBOL, por ejemplo, la proposición "Si-entonces-sino" no puede ser implementada directamente, pues no permite el uso adecuado de proposiciones anidadas como se muestra en la figura 3.4.1. La desventaja de éste hecho es que el código de un módulo no permanece unido, sino que hay que dividirlo en secciones y lugares físicos diferentes.

```
ESTRUCTURA                          IMPLEMENTACION COBOL

Si P entonces                       IF P THEN
   Si Q entonces                       PERFORM IF-INTERNO
      Instrucción-1                    INSTRUCCION-3
   Sino                             ELSE
      Instrucción-2                    INSTRUCCION-4
   Fin (si)
   Instrucción-3                    IF INTERNO SECTION
Sino                                   IF Q THEN
   Instrucción-4                       INSTRUCCION-1
Fin (s).                            ELSE
                                       INSTRUCCION-2
```

FIGURA. 3.4.1 PROBLEMA DEL USO DE LA ESTRUCTURA SI-ENTONCES-SINO ANIDADAS EN COBOL.

En el caso de los compiladores FORTRAN 1966 [FOR66], el mayor problema consiste en que solo reconocen instrucciones simples y por lo tanto, la estructura lógica del programa deber ser construida utilizando la instrucción de ramificación incondicionada ("GO TO"), la cual, si se utiliza indisciplinadamente obscurece la lógica del programa. En cualquier lenguaje de programación deberemos restringirnos a usar instrucciones que, aunque no sean traducciones directas del pseudocódigo, al menos si sean implementaciones ordenadas o "emulaciones" de las estructuras básicas.

A continuación se presentan dos técnicas de codificación: Las emulaciones y los precompiladores, especialmente útiles cuando se utiliza un lenguaje de programación que no cuenta entre sus instrucciones normales: con las estructuras básicas, definidas en la sección de Técnicas de Diseño, específicamente en el punto de programación estructurada (2.3.3. Para terminar se menciona una tercera técnica que puede aplicarse a cualquier tipo de lenguaje y se conoce como estilo de programación.

3.4.1.1 Emulaciones -

A continuación se presentan, en FORTAN 66, las emulaciones de las principales estructuras básicas definidas en la programación estructurada. (Véase figura 3.4.2)

ESTRUCTURA (SI-ENTONCES-SINO)

```
Si P entonces                       IF (.NOT.P) GO TO N1
   Instrucción-1                       Instrucción-1
                                    GO TO N2
Sino                             N1 CONTINUE
   Instrucción-2                    Instrucción-2
Fin(si)                          N2 CONTINUE
```

ESTRUCTURA (MIENTRAS)

```
Mientras (condición)              NI  IF (.NOT.condicion) GO TO
N2
   Instrucción-1                        Instrucción-1
                                         GO TO  N1
Fin(mientras)                     N2  CONTINUE
```

ESTRUCTURA (EJECUTA)

```
Ejecuta                                GO TO N2
                                  N1   IF (condicion) GO TO N3
                                  N2   CONTINUE
   instrucción-1                       Instrucción-1
                                        GO TO N1
Hasta (condición)                 N3   CONTINUE
                                        END IF
                                  N2   CONTINUE
```

8

ESTRUCTURA (REPITE)

```
Repite inicial,final,incr.              INDICE= INICIAL
                                    N1  CONTINUE
   Instrucción-1                          Instrucción-1
                                        INDICE=INDICE+INCR
Fin(repite)                             IF(INDICE.LE.FINAL)GO
                                                       TO N1
```

FIGURA 3.4.2 EMULACIONES EN FORTRAN 66 DE ESTRUCTURAS BASICAS

Las desventajas de la emulación de las estructuras básicas son: primero, que el número de pasos de descomposición necesarios para llevar el planteamiento original del programa a su implementación es mayor [CIM 81], y segundo, que la claridad de un programa escrito en pseudocódigo se pierde al traducirlo a código [JEN 79].

Tomando en consideración los puntos expuestos anteriormente, la solución más popular para la emulación de las estructuras básicas es el uso de un precompilador.

9

3.4.1.2 Precompiladores.-

Un precompilador es un programa de computadora que extiende la síntaxis de un lenguaje de programación específico permitiendo al programador codificar sus programas utilizando las estructuras básicas (si-entonces-sino, mientas, etc.) en combinación con las instrucciones propias del lenguaje.

El uso de precompiladores presenta varias ventajas sobre el intento de programar en forma estructurada en lenguajes no estructurados:

a. Portabilidad

b. Menores violaciones a las reglas de programación estructurada.

c. Mayor aproximación entre los procesos de diseño y codificación.

d. Mayor confiabilidad.

e. Mayor mantenibilidad.

Algunos de los precompiladores más populares son el META COBOL [WEI 77], WATFOR [KER 76] y HIFTRAN [HIF 76].

En la práctica la programación estructurada es una herramienta que permite escribir programas más claramente concebidos, más legibles y por lo tanto, con menos errores; sin embargo, no puede afirmarse que el mero uso o emulación de las estructuras de control básicas lleve automáticamente a escribir programas estructurados.

3.4.1.3 Estilo Del Programador -

La claridad de un programa depende en gran medida del estilo del programador. Si bien los principios de los que constituye un buen estilo de programación no pueden tampoco ser expresados como reglas mecánicas de la buena programación, la experiencia nos indica algunas normas que pueden servir como "guía de estilo" y las cuales se detallarán en la sección de Normas de Codificación (4.2.3.).

Resumiendo, la mejor documentación de un programa la constituye la claridad de su estructura; además como ya se mencionó la documentación más confiable de un programa es el programa mismo, pues solo leyendo el código se puede detectar lo que hace el programa. Por lo tanto se recomienda la selección de lenguajes que estimulen una programación legible y estructurada, evitando la necesidad de instrucciones de flujo arbitrariamente complicadas.

3.4.2 Técnicas De Integración.-

En esta sección se presentan algunos tipos de pruebas recomendadas para identificar los errores de cómputo que se originan en la fase de integración del proceso de programación.

En la descripción de la fase de integración se describieron dos alternativas para validar módulos de programación: integración incremental y no incremental.

Asociadas a la integración incremental existen dos variantes en el desarrollo de las pruebas:

Pruebas de arriba hacia abajo (PARAB)

Pruebas de abajo hacia arriba (PABAR)

Antes de pasar a explicar las diferencias entre éstos dos tipos de pruebas en el contexto de un esquema de integración incremental, es conveniente señalar algunos términos que son utilizados comúnmente en forma errónea:

Los términos pruebas de arriba hacia abajo (PARAB), desarrollo de arriba hacia abajo y diseño de arriba hacia abajo son utilizados como sinónimos.

Pruebas (PARAB) y desarrollo de arriba hacia abajo si son efectivamente sinónimos, ya que representan un ordenamiento en la secuencia de codificación y pruebas modulares.

Diseño de arriba hacia abajo es algo distinto. Un programa de computadora originalmente diseñado de arriba hacia abajo puede ser ensamblado incrementalmente de arriba hacia abajo o de abajo hacia arriba.

Pruebas de abajo hacia (PABAR) se interpretan usualmente en forma equivocada como pruebas no incrementales, debido a que éstas comienzan en forma identica (cuando los módulos del nivel jerárquico más bajo son validados). Pero como aquí se señala, pruebas de abajo hacia arriba (PABAR) es una alternativa de la estrategia incremental de la fase de integración.

La estrategia 'PABAR' comienza con los módulos del nivel jerárquico más bajo dentro de un diagrama de estructura es decir, con aquellos módulos terminales que no requieren llamar otros modulos. (módulos 'B','C', y 'D' de la figura 2.4.1.). Una vez validados éstos módulos terminales no existe una estrategia óptima que permita definir el camino a seguir. El único requerimiento de las pruebas en el nivel jerárquico superior es la obligación de haber validado todos y cada uno de los módulos subordinados al módulo seleccionado.

Para validar 'A' será necesario haber probado 'B', 'C' y 'D' en el diagrama de estructura de la figura 2.4.1, que con lleva la necesidad de eleborar módulos 'direccionadores' que contengan casos de prueba que especificamente validen entradas y salidas de cada módulo.

La estrategia 'PARAB' comienza con el módulo principal del programa de computadora. Para probar este primer módulo será necesaria diseñar módulos 'ciegos' que simulen las funciones de los módulos subordinados. (para el caso de la figura 2.4.1 hubiese sido necesario diseñar módulos 'ciegos' para 'B', 'C', y 'D'), una tarea que en general resulta más complicada que la generación de módulos 'direccionadores'.

La función de los módulos 'ciegos' es generalmente mal interpretada, ya que no debe ser su función alarmar 'llegue a la subrutina B', o simplemente regresar el control al nivel jerárquico superior, sin antes simular su funcionamiento.

Una ventaja asociada con 'PABAR' es evitar la posibilidad de traslapar las fases de diseño y pruebas, ya que no debe comenzar la etapa de pruebas hasta no haber diseñado el último de los módulos del programa.

Asimismo, el problema de dejar inconclusas las pruebas de un módulo, comenzando las de otro, debido a las diferentes funciones por simular y codificar en los módulos 'ciegos', desaparece en 'PABAR'.

Los problemas asociados con la imposibilidad o dificultad de generar casos de prueba dentro de una estrategia 'PARAB', no existen en las pruebas 'PABAR', ya que se asocia un caso de prueba a cada módulo 'direccionador'. Las pruebas se imponen directamente a los módulos subordinados, evitando la necesidad de contemplar otro módulos del programa de computadora.

En conclusión, podríamos señalar que la complejidad en el diseño de los casos de prueba de la estrategia 'PARAB' y la disponibilidad de herramientas de prueba para simplificar los módulos de prueba 'direccionadores' orillan a respaldar la técnica 'PABAR' para la integración incremental de programas de computadora.

3.5 Técnicas De Revisión

El proceso de desarrollo de programas en sus diferentes fases, requiere de revisiones formales al final de cada etapa del desarrollo. Estas revisiones se conducen con el objeto de permitir al cliente, usuarios, diseñadores y administradores del proyecto la evaluación del progreso logrado.

Una de las razones que justifican el proceso de 'revisión' a lo largo del desarrollo de programas es que se ha comprobado que las revisiones aumentan la confiabilidad del producto final. La confiabilidad de un programa de computadora se puede definir como la probabilidad de que el programa opere por un periodo de

tham, MA, 1973.

[10] S. Hori, "CAM-I long range planning final report for 1972," IITT Res. Inst., Dec. 1972.

[11] L. J. Peters, *Handbook of Software Design*. New York: Yourdon Press, 1980.

[12] S. L. Pollack, H. T. Hicks, Jr., and W. F. Harrison, *Decision Tables, Theory and Practice*. New York: Wiley-Interscience, 1971.

[13] M. Montalbano, *Decision Tables*. Palo Alto, CA: Science Research Associates, 1974.

[14] M. Hamilton and S. Zeldin, "Top-down, bottom-up, structured programming and program structuring," Charles Stark Draper Lab., Massachusetts Institute of Technology, Cambridge, MA, Document E-2728, Dec. 1972.

[15] J. Rood, T. To, and D. Harel, "A universal flowcharter," in *Proc. NASA/AIAA Workshop on Tools for Embedded Computer Systems Software* (Hampton, VA), pp. 41-44, Nov. 1978.

[16] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, Aug. 1973.

[17] N. Chapin, R. House, N. McDaniel, and T. Wachtel, "Structured programming simplified," *Comput. Decisions*, pp. 28-31, June 1974.

[18] N. Chapin, "New format for flowcharts," *Software Practice Experience*, vol. 4, pp. 341-357, 1974.

[19] P. G. Hebalkar and S. N. Zilles, "TELL: A system for graphically representing software design," IBM Res. Rep. RJ 2351, Sept. 1978.

[20] T. DeMarco, *Structured Analysis and System*. New York: Yourdon, 1978.

[21] J. D. Warnier, *Logical Construction of Programs*. New York: Van Nostrand Reinhold, 1977.

[22] M. A. Jackson, *Principles of Program Design*. London, England: Academic Press, 1975.

[23] R. C. Linger, H. D. Mills, and B. F. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[24] H. F. Ledgard, "The case for structured programming," *Nordisk Tidskrift for Informations Behandling*, Sweden, vol. 13, pp. 45-47, 1973.

Software Quality Assurance: Testing and Validation

JOHN B. GOODENOUGH AND CLEMENT L. McGOWAN

Abstract—There are many pitfalls for the unwary hardware engineer who must develop software. In particular, hardware quality control procedures and concepts can easily be misapplied. The purpose of this paper is to help hardware-oriented engineers apply some of the quality assurance lessons learned by software engineers. We first discuss how software is and is not analogous to hardware, and then outline recommended software engineering approaches for developing high-quality software products. We concentrate on methods for preventing and detecting software errors.

I. Introduction

A. The Software-Hardware Analogs

THERE ARE many similarities between software and hardware design and development, but unless one is careful, the similarities can be misleading. For example, hardware failures are sometimes considered analogous to software failures, but often the wrong analogies are drawn. In particular, component deterioration is the usual cause of hardware failure. In contrast, software failures are almost always *design errors* that show up only when the software is used under certain conditions.

For example, a hand calculator was once manufactured that did not correctly compute the sine function for all argument values. All units computed the same incorrect value for particular arguments. This was clearly a design error—the circuit for calculating sines was incorrectly designed. Since the calculator gave correct answers for most arguments, the problem was only discovered by a few users. This is typical of design errors, which are latent until the hardware or software is exercised under the appropriate conditions.

Because software errors are analogous to hardware design errors, software quality assurance focuses on techniques for getting the design right. Adapting hardware quality assurance procedures to software development means focusing on the procedures used to prevent and detect design errors. This is the proper analogy between software and hardware quality assurance. It illustrates our point that unless one makes the proper analogies, intuitions about how to ensure hardware quality will lead one astray when working with software.

A comparison of hardware and software life cycles is given in Fig. 1. This figure indicates the phases of developing a new hardware or software product. Although Fig. 1 provides an

Manuscript received February 12, 1980; revised April 14, 1980.

The authors are with SofTech, 460 Totten Pond Road, Waltham, MA 02154.

0018-9219/80/0900-1093$00.75 © 1980 IEEE

	HARDWARE	SOFTWARE
DEVELOPMENT	Determine User Requirements	Determine User Requirements
	Develop Product Concept (Functional Design)	Develop Product Concept (Functional Design)
	Specify Component Design (Detailed Design)	Specify Component Design (Detailed Design)
	Build and Test Prototype	*Implement and Test Programs
	Develop Manufacturing Techniques	--
INSTALLATION	Manufacture Product	*Copy Programs
	Make Product Available to User	Make Program Available to User
MAINTENANCE/ IMPROVEMENT	Maintenance (Correct Component Failures)	--
	Recall Product to Correct Design Flaws	*Maintenance (Correct Implementation and Design Errors)
	Enhance Product	Maintenance (Provide Enhanced Capabilities; Adapted to Changed User Environment)
PHASE-OUT	Unit is Unusable and Unrepairable (replace with new unit)	--
	Product is Obsolete	Product is Obsolete

*marks important differences in terminology

Fig. 1. Corresponding steps in software and hardware product life cycles.

oversimplified and idealized description, it shows clearly that similar terms in the two fields sometimes have radically different meanings. Perhaps the most important differences are

1) coding programs is not equivalent to manufacturing a product;
2) maintenance refers to quite different processes (see Lehman [16] for further discussion of this point);
3) program development and test is conceptually similar to developing and testing a hardware prototype, but in software, the "prototype" is the first system that gets delivered to users.

The focus of this paper is software testing and quality assurance. As Fig. 1 indicates, we will therefore be discussing procedures and concepts aimed at producing a high quality *design* (in the hardware sense). Testing and quality control procedures used in manufacturing or maintaining a hardware product are not directly relevant to software quality assurance.

Finding design errors is difficult for both hardware and software engineers, and finding such errors is more difficult as the complexity of the product increases. In general, software products are equivalent to very complex hardware products, so it is no wonder developing high-quality software is difficult and costly.

B. *Sources of Software Errors*

The basic steps in developing a software system are

1) defining user requirements;
2) deciding what functions and major components a system must provide to meet these requirements;
3) designing and specifying the intended behavior of individual software components;
4) implementing (i.e., coding) software components.

Each of these software development activities is subject to error:

1) construction errors—failure of software components, as implemented, to satisfy their specifications;
2) specification errors—failure to accurately specify the intended behavior of a unit of software construction;
3) functional design errors—failure to establish an overall design able to meet identified requirements;
4) requirements errors—failure to identify user needs accurately, including failure to communicate these needs to software designers.

Different quality assurance techniques are required to deal with these errors, since they arise at different stages of software development. In this paper, we will concentrate on design, specification, and construction errors.

Software is always tested before being released for operational use and testing is typically the last activity before release. This gives testing high visibility to developers and users and often leads to an undue reliance on tests as the means of ensuring software quality.

Since tests are performed only after code is written, they are a costly method of detecting errors, especially specification and design errors. Correcting these errors may require discarding some software that has already been written and tested. A cost effective approach to reducing software errors must attempt to prevent and detect errors as soon as possible. Ideally, software design errors are detected during the design phase, before specifications of software components are written, and inconsistencies between specifications and the design are detected before code is written.

In Section II, we will discuss principles and practices underlying the development of software tests, and in Section III, we will discuss an integrated approach to software quality assurance. The integrated approach discusses what quality assurance procedures and principles should be applied at each phase of software development, rather than focusing solely on the development and evaluation of test cases.

II. Software Testing Principles

From an engineering point of view, software testing has traditionally been *ad hoc*, with few principles and no theoretical work indicating how to construct an adequate set of tests or how to measure the adequacy of a set of tests that someone has constructed. In the last few years, however, some theoretical work has increased our understanding of how to construct tests. We will discuss this recent work to show how the basis for an engineering approach to software testing is developing.

Software testing involves the execution of programs with selected inputs, called test cases. The results of test executions are then used to decide whether the program is operating acceptably.

The most common objective in software testing is to determine whether a program is correct, i.e., whether the program produces specified outputs when presented with permitted inputs. Although correctness may at first seem to be the most important property a program can have, this is by no means the case, particularly for large software systems. (See [20] for a more detailed discussion.) Large programs are often so complex they never completely satisfy their specifications, and yet, they may be quite usable because failures are encountered infrequently in practice, and when they do occur, their impact on a user is acceptably small. Hence correctness is not necessary for a program to be usable and useful. Nor is correctness sufficient. A correct program may satisfy a narrowly drawn specification and yet not be suitable for operational use because in practice, inputs not satisfying the specification are presented to the program and the results of such incorrect usage are unacceptable to the user. If a program is correct with respect to an inadequate specification, its correctness is of little value.

Consequently, although testing for correctness is the most common and best understood testing goal, correctness is by no means the only important property of usable software—reliability, robustness, efficiency, and other properties (see [21]) are also of significant importance. But these properties are less commonly the focus of testing activities.

In designing correctness tests, it is important to keep a few principles in mind.

1) "Black box" tests (i.e., test cases chosen without knowledge of how a program has been implemented) cannot ensure that all correctness errors are detected unless the tests are exhaustive, i.e., consist of all inputs in the program's input domain. Exhaustive tests are almost never practicable, however. Input domains are just too large; often they are not finite.

2) Even "glass box" tests (i.e., tests chosen with full knowledge of how a program has been implemented) are not necessarily adequate to detect all correctness errors. For example, selecting tests so all branch conditions in a program or even all execution paths through a program are exercised will not detect an error if a branch or path that should be present in the program is missing (see [12]).

3) The most effective way to design tests is to hypothesize certain software errors and then to select test cases that will fail if the errors are present. We discuss aspects of this approach below.

Current research approaches for developing correctness tests fall in two catagories—deterministic and probabilistic. Deterministic methods select tests that will fail if certain kinds of errors (and only those kinds of errors) are present in a program. Probabilistic methods provide estimates of the likelihood of undetected errors remaining in a program without ever fully guaranteeing that all errors (of certain kinds) have been eliminated. Probabilistic methods are aimed not so much at defining how to select test data but rather at evaluating the effectiveness of tests in uncovering errors. An ideal software testing method gives both a reliable mea[illegible] effectiveness and, if the measure indicates more testing is needed, shows a tester where further testing effort will be profitable.

For example, the deterministic approach is illustrated by Chow's [6] technique for testing communications software protocols and programs implementing them. Such protocols can be described in terms of finite state machines. Chow has shown that given an upper bound on the number of states in the correct finite state machine, a set of test cases can be generated that will detect all errors due to missing states and missing or incorrect state transitions. A program implementing the protocol can be deterministically tested with these inputs to determine if it has realized the intended design, i.e., successful execution implies the implementation is correct. West [31] describes a similar procedure for detecting system deadlocks, potential losses of messages, and other communications protocol errors. West also requires that the protocols be modeled as finite state machines.

Another example of the deterministic approach is the one developed by White and Cohen [32]. They attempt to detect only errors due to incorrectly written branch conditions in a program. They partition the input domain of a program into equivalence classes such that each element of a class causes execution of the same program control flow path. Their testing strategy describes how to select test data to determine if the boundaries defining a class have been shifted from their correct position. For example, given a boundary defined by an ordering relation, e.g., $3X + 2 > Y$, they show why in general three test cases are needed to show whether the boundary has been correctly specified—two test points on the boundary and one off the boundary; e.g., (3, 11), (-1, -1), and (1, 4). A more naive approach to test case selection would assume that two test cases would suffice—one in which the relation was satisfied, and one in which it was not. But Cohen and White's analysis shows why two cases are insufficient to detect errors in the coefficients. Showing how intuitive testing approaches can be inadequate is an important result of deterministic testing research.

The common elements in deterministic testing approaches are 1) their focus on detecting only certain kinds of errors in the absence of other kinds of errors, and 2) their development of methods for selecting test data that is guaranteed to detect these errors if they are present. A productive area for further research is defining test case selection methods for additional classes of errors. Such research is needed to provide a firm theoretical basis to guide software engineering practice

In constrast to the deterministic approach to testing, in which one attempts to find test cases guaranteed to detect certain kinds of errors, DeMillo *et al.* [8] have developed a probabilistic method for assessing the quality of a test set without knowing precisely what errors the test set is able to detect. Their method requires "mutating" (i.e., modifying) the program to be tested by introducing small changes that are likely to be errors. The original program and the mutated programs are tested using the same set of test cases. The quality of the test set is determined by the number of mutants generated and by how many fail to pass the tests. Ideally, all mutants fail. When mutants are not eliminated by the tests, one must either attempt to find test data for which the mutants will fail or demonstrate that the mutants are equivalent to a correct program.

The principle underlying the mutation approach to testing is "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [8].

There is no certainty that all errors are discovered if the set of tests eliminates all incorrect mutants. The assumption, however, is that in practice any remaining errors are not important. Research is currently underway to see to what extent this assumption holds and to see what kinds of program mutations are most useful for validating test sets [1].

Software testing is not currently an engineering discipline. The theoretical basis needed for such a discipline simply does not yet exist. But our examples of new testing approaches show that some progress is being made.

While waiting for theory to catch up with the needs of software developers, today's software engineers must apply rules of thumb learned by experience. These rules are well explained by Myers [23], [24], whose books contain what is probably the best currently available collection of good practices for developing software tests.

Perhaps the most important thing to remember about software testing is that although it is necessary, it cannot be the sole means of ensuring software is reliable, robust, and useful. To obtain software having these properties in sufficient measure, one must use appropriate quality assurance techniques at each stage in the software development process. Testing is only one of these stages.

III. An Integrated Approach to Quality Assurance

As noted previously, software development is analogous to the building and testing of a hardware prototype. Accordingly, attempts to assure software quality should attend principally to the design process and should analyze its outputs. Indeed, H. Mills [18] recommends that

> Given double the budget and schedule (to test the sincerity of a requirement for ultrareliability), do not spend the extra on testing—spend it in design and inspection.

In this section, we review the software development process's major activities, namely design, construction, and testing, from the perspective of what proven quality assurance principles and procedures should be applied. Coverage of all the applicable techniques and tools is not possible here. Rather, the intention here is to survey the state of the art as practiced, with suitable references supplied for reader follow up.

A. Reducing Design and Specification Errors

Software design can be characterized as allocating requirements to the components of an architecture. This characterization stresses that a design consists of parts (modules) and their interconnections (interfaces) for the purpose of realizing a given set of requirements. Clearly this presupposes that the software requirements are defined and analyzed prior to the design activity. Without first satisfying this important presupposition all subsequent efforts to assure a quality product are, at best, misguided.

Presently, there are quite a few competing "design methodologies" offering a software designer (or a design team) prescriptions of explicit steps to follow as well as a specific graphic notation for representing the resultant design. We mention here some of the more prominent approaches: structured design [21], [22], [29], [36], the Jackson method [15], the Warnier-Orr approach [25], [30], the structured analysis and design technique (SADT) [9], [27], the requirements engineering validation system (REVS) [2], the systematic design methodology (SDM) [3], [14], the operational software concept [26], [35], higher order software (HOS) [13], and the architecture definition technique (ADE) [4].

Generally, by using a top-down strategy, each of these methods develops a software system design as a group of communicating modules. Such top-down strategies proceed from the general to the particular by first defining the context, the function, and the interfaces for any module before specifying the internal details of that module. However, the various methods are fundamentally different in what they regard as important and in how they attack a design problem. Naturally, the graphic notation associated with a particular method is well suited to representing "important" system aspects. That is, the very notation for representing a design architecture enforces abstraction by excluding or deferring less important aspects (as judged by a method's creators). Some of the system aspects that have been used by various design methods for creating a "good" design are: control flow, data flow, the input and output data structures, the internal data-base structure, stimulus/response threads through the system, hierarchical decomposition of system functions, system states (or modes), and the transitions between them, dependencies between functions, and major system features that should be easily changeable.

For a software quality assurance viewpoint, there are many advantages to adopting a particular design method and graphic notation for representing a design. Such a standard establishes a basis for training and for project communication. This eases the introduction of new personnel during development and provides a useful "mental framework" for the eventual maintainer. Once selected, design standards enhance the efficacy of all internal design reviews by providing some objective criteria that can be uniformly applied when assessing design "qualities" such as completeness, correctness, and clarity. Moreover, standard design documentation can—and should—be supported by software tools that store the design information for subsequent queries, that automatically produce a standard formatting, and that report on the consistency of the design information entered thus far. Some research and development efforts (based on particular design methods) have extended the design support tools to include simulations or "animations" of the currently stored design. In summary, with respect to using a design method experience indicates that any reasonably systematic strategy is better than none.

Besides design reviews internal to the development team, several formal design reviews for the benefit of senior management, the customer, and an independent quality assurance group are advisable. Established practice now holds two such formal exercises called the preliminary design review (PDR) and the critical design review (CDR). The PDR assesses the proposed design architecture for logical and technical feasibility. All of the software components (both procedural and data) should be functionally specified with interfaces identified in some detail. In addition, a plan scheduling subsequent design, implementation, testing, and integration tasks must be provided. Draft user and maintenance manuals and a proposed acceptance test plan are appropriate. The CDR focuses on implementation and performance feasibility. It typically requires the detailed algorithms, data structures, and interface formats along with memory and execution time budgets for each software elements. Usually satisfying the CDR "action items" is the prerequisite to initiating actual coding. With respect to internal and formal design reviews Glass [11] has said the following.

> Most important, the success or failure is dependent on people. The people who attend must be intelligent, skilled, knowledgeable,

in the specific problem area, cooperative, and have the time available to dedicate themselves to the review. Only the interactions of capable people can make a design review successful.

The quality assurance role in a design review is to examine the design for completeness (are all requirements addressed?), for quality (using objective criteria derived from the selected design method), and for adherence to standards.

B. Reducing Construction Errors

Software construction is concerned with the design of module details and then with expressing these details in executable code. Algorithm design, data layout, and access considerations are central to this activity. Already there is a rich body of knowledge about specific algorithms and data structures as well as their relative performance tradeoffs. By demonstrably helping to improve the quality of software construction, two general practices have gained widespread acceptance—structured programming for detailed routine and data design and a higher order language (HOL) for coding programs.

Structured programming involves the hierarchical fabrication of programs and data structures by means of the repeated application of some basic composition rules. For example, in "structured coding" basic statements (such as arithmetic evaluation and assignment) may be combined into compound statements only by using statement sequencing, conditional selection of a group of statements to be executed, or conditional iteration of some statements. That is, the unconditional branch or GO TO statement is excluded from structured coding (sometimes erroneously characterized as GO TO-less programming). These restrictions generally simplify a routine's flow-of-control logic and have demonstrably reduced a significant source of coding errors. In less than a decade, structured programming has gone from a proposed approach [7], [19], [33] to an operational standard for most major software developers.

Coding standards incorporate aspects of structured programming with rules for indentation (to reflect nesting), header comments, identifier names, and the length of routines (to encourage short, functional procedures). A major premise of structured programming is that programs are to be read by people as well as by computers. Proceeding on this premise, quality assurance activity has recently included code reading. The traditional desk checking by the programmer is profitably augmented with a more formal peer code review [10] that is scheduled, conducted using checklists for guidance, and tracked with all the trappings of reports, action items and follow ups.

Using an HOL makes software construction much more reliable and productive than using assembly language. Good optimizing compilers can now produce code that is within 25 percent in both space and execution time of well-crafted assembly language. HOL code is so much easier to produce, checkout, maintain, and modify that it should be used for all but the most stringently constrained target environments. Indeed, Fred Brooks claims, "I cannot easily conceive a programming system I would build in assembly language" [5]. Interactive debugging in the symbolic terms of the HOL source program substantially speeds routine check out. Modern typed HOL's (such as Pascal [34]) catch many data interface and misuse errors during compilation. In some cases, detailed design can be better represented in a suitable source HOL (because of the automatic check out and cross referencing) than in flowcharts or in some program design language (PDL)

C. Extending Testing Principles to Software Systems

The burden of producing detailed design documentation and of unit testing various routines often obscures what the essential product of software development is. The real goal should be to place a system (including people, machines, and software) into satisfactory operation. Consequently system and software integration are prominently placed on the critical path. To better reduce and control the associated risk, software integration and testing efforts should be distributed over as much of the coding effort as possible. Indeed, a more accurate measure of current project development status is "what percentage of the total number of software modules identified during architectural design have been coded and integrated into an operational subsystem?" Such a measure is far superior to the more frequently used "how many lines have been coded?" or "how many modules have been unit tested?" Clearly a percentage-of-modules-integrated "measure" dictates using some variant of a top-down implementation and testing sequence [17]. A significant strategy is to identify some operational subsets (or "builds") of the intended system and to develop and deliver in succession these subsets [28]. This approach provides an early check out of the major software and user interfaces while helping to identify some important but nontechnical potential problems (e.g., user training, delivery format and mechanism, coordination of multiple contractors, etc.). In fact, testing should drive development in the sense that plans for testing software subsets should determine the implementation sequence.

Testing can be regarded as gathering information on the software's reliability. Thus viewed, test requirements, plans, and procedures as well as design reviews are as much a part of testing as is exercising software on a machine. It is beneficial to have a separate project test or quality assurance team address these issues. For example, a test procedure must clearly define the series of actions required to verify that a product meets its requirements. These actions may include

1) configuring—arranging the software, hardware, people, and logistics for the test;
2) conditioning—bringing the system to the initial state for testing;
3) introducing data—entering either "live" or simulated data;
4) starting the test—initiating and synchronizing where necessary;
5) collecting data—gathering snapshot, frequency, and resource utilization information;
6) displaying results—selecting and formatting results as directed;
7) analyzing results—comparing actual to expected.

A variety of software tools assist in conducting test procedures. These include environment simulators, test data generators, and test coverage analyzers.

In summary, software development and test is fairly analogous to the development and testing of a hardware prototype; design errors predominate. This insight helps in identifying some major sources of software errors. A serious software quality assurance effort that focuses on these error sources must be active from a project's inception. The issues of requirements testability, traceability, and cost must be resolved early. Design reviews must be conducted for the purpose of finding design errors sooner rather than later. Proven methods for designing, programming, making the evolving product visible, and controlling different system configurations can be standardized and then supported by a suitable software project library with associated procedures. Of course, execution testing remains the principle means for determining the current status of a software product.

REFERENCES

[1] A. T. Acree, R. A. DeMillo, T. J. Budd, R. J. Lipton, and F. G. Sayward, "Mutation analysis," NTIS Rep. ADA-076575, Sept. 1979.

[2] M. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans Software Eng.*, vol. 3, pp. 60-68, Jan. 1977.

[3] R. Andreu, A systematic approach to the design and structuring of complex software systems, Ph.D. dissertation, Sloan School of Management, Massachusetts Institute of Technology, 1978.

[4] C. Bachman and J. Bouvard, "Architecture definition technique its objectives, theory, process, facilities and practice," in *Proc. ACM SIGFIDEF Data Description Access and Control*, pp. 257-305, 1972.

[5] F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[6] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans Software Eng.*, vol. 4, pp. 178-186, May 1978.

[7] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* New York: Academic Press, 1972.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Comput.*, vol. 4, pp. 34-43, Apr. 1978.

[9] M. Dickover, C. McGowan, and D. T. Ross, "Software design using SADT," in *Proc. Nat ACM Conf.*, pp. 125-137, 1977.

[10] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J*, vol. 15, pp. 182-211, 1976.

[11] R. Glass, *Software Reliability Guidebook* Englewood Cliffs, NJ: Prentice-Hall, 1979.

[12] J. B. Goodenough, "A survey of program testing issues," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 316-340.

[13] M. Hamilton and S. Zeldin, "Higher order software—A methodology for defining software," *IEEE Trans Software Eng.*, vol. 2, pp. 9-32, June 1976.

[14] S. Huff and S. Madnick, An extended model for a systematic approach to the design of complex systems, NTIS Rep. ADA-056565, 1978.

[15] M. A. Jackson, *Principles of Program Design* New York: Academic Press, 1975.

[16] M. M. Lehman, "Programs, programming, and the software life cycle, this issue, pp. 1061-1076.

[17] C. L. McGowan and J. R. Kelly, *Top-Down Structured Programming Techniques* New York: Van Nostrand, 1975.

[18] H. Mills, "Software development" in *Research Directions in Software Technology*, P. Wegner Ed. Cambridge, MA: M.I.T. Press, pp. 87-105.

[19] ——, "Top-down programming in large systems," in *Debugging Techniques in Large Systems*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1971, pp. 41-55.

[20] J. D. Musa, "The measurement and management of software reliability," this issue, pp. 1131-1143.

[21] G. J. Myers, *Reliable Software Through Composite Design*. New York: Van Nostrand, 1975.

[22] ——, *Composite/Structured Design* New York: Van Nostrand, 1978.

[23] ——, *Software Reliability Principles and Practices*. New York: Wiley, 1976.

[24] ——, *The Art of Software Testing* New York: Wiley, 1979.

[25] K. Orr, *Structured Systems Development*. New York: Yourdon Inc., 1977.

[26] J. Rodriguez and S. Greenspan, "Directed flowgraphs: The basis of a specification and construction methodology for real-time systems," *J. Syst. Software*, vol. 1, pp. 19-27, Jan. 1979.

[27] D. Ross, "Structured analysis: A language for communicating ideas," *IEEE Trans. Software Eng*, vol. 3, pp. 16-33, Jan. 1977.

[28] D. Schultz, "A case study in system integration using the build approach," in *Proc. Annu ACM Conf.*, pp. 143-151, Oct. 1979.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, pp. 114-139, 1974.

[30] J. Warnier, *Logical Construction of Programs* Leiden, Germany: Stenfert-Kroese, 1974.

[31] C. H. West, "General technique for communication protocol validation," *IBM J. Res. Develop.*, vol. 22, pp. 393-404, 1978.

[32] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans Software Eng.*, vol. 6, pp. 247-257, May 1980.

[33] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol 14, pp. 221-227, Apr. 1971.

[34] ——, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35-63, 1972.

[35] K. Wong and J. Engelland, "Operational software concept: A new approach to avionics software," in *Proc. AIAA Digital Avionics System Conf.*, 1975.

[36] E. Yourdon and L. Constantine, *Structured Design* New York: Yourdon Inc., 1975.

INSIDE:

History, Why Software Testing?, Organizing for Testing, Psychology of Testing, Management Guidelines

An Overview of Software Testing

PAYOFF IDEA. Dramatic improvements in hardware price/performance during the last two decades have increased pressure on software developers to keep up with these advances. The current emphasis on techniques and tools for decreasing development time and for improving system quality in this labor-intensive area is an indication of this pressure. DP managers need only consider the percentage of their budgets allotted to development and maintenance as well as the amount of effort wasted because of system failures to realize the seriousness of the software problem. An effective software testing program can significantly improve the quality of software by reducing errors and preventing system failures. This portfolio provides the DP manager with an overview of testing and presents guidelines for setting up a testing function.

PROBLEMS ADDRESSED

Of all the issues in computer science and data processing, none has received more attention in the last decade than software quality. Software quality underlies many of the problems facing the DP community, often determining whether particular processing solutions are acceptable. Many systems have failed to meet minimum levels of acceptability, while some (perhaps too few) have met with genuine success.

What makes the difference between success and failure in software systems? The answer to this question is manifold: it can include the application of good software engineering techniques, the use of modern programming practices and software design methodologies, and the use of more reliable high-level languages. The methods used to ensure the quality of the delivered system, however, must also be considered. Quality is ensured primarily through some form of software testing. (There are a few instances of formal program verification, but the total amount of software processed with this theoretically important but costly method is minimal.) Software testing may not be called that at all: advanced debugging, interface testing, acceptance testing, or similar terms are often used.

There is growing emphasis on the importance of testing as a quality-assurance discipline for software. This portfolio is the first in a series of three that discusses software testing. This portfolio outlines the history of software testing, the reasons for testing, organizing for testing, and management guidelines for developing a cost-effective testing function. The second portfolio (4-04-02) describes testing tools currently available, and the third (4-04-03) discusses testing methodologies.

HISTORY OF TESTING

The first formal technical conference devoted to software testing was held in June 1972 at the University of North Carolina. Subsequent to that meeting several international software engineering conferences have been held as well as many smaller workshops and symposia devoted either fully or in part to program testing. At the Oregon Conference on Computing in the 1980s, testing technology was considered a formal part of software engineering technology.

The history of testing actually dates to the beginnings of computing: programs that ran on the earliest machine were tested. Since then software quality has varied widely; as might be expected, the failures are often more renowned than the successes. The famous missing comma in the Venus probe software and the structural faults that led to the detection of the moon as an incoming ICBM have reached the public as examples of how even the most sophisticated software can fail.

The need for effective software quality assurance at affordable prices is steadily increasing. This portfolio includes practical guidelines for developing a cost-effective testing function.

Software testing has one major psychological obstacle to widespread acceptance: a negative reward structure. The objective of a dedicated software tester is to find and report errors; this attitude may conflict with the goals of the software implementors and their managers. Thus, in most cases, it is necessary to consider a special organizational structure to isolate this negativity while enabling the two functions to work together effectively.

As more experience is gained with testing, an increasingly strong set of guidelines will be developed for handling the software quality-assurance problem. For now, knowing how to maximize the benefits of current techniques for achieving software quality is the responsibility of the well-informed quality-assurance manager.

WHY SOFTWARE TESTING?

This section describes the basics of testing, suggests some of the primary motivations for using formal testing methods, and discusses some of the benefits of testing.

The Basics of Testing

Testing a computer program entails three basic steps:

- Running the program with a controlled set of inputs
- Observing the effect of inputs on program runtime
- Examining the program outputs to determine their acceptability

These steps are followed informally during the debugging process; formal program testing is more concerned with showing precisely what the program does. One way to distinguish between formal testing and debugging is to examine the tools used. Debugging relies on program production tools (e.g., compilers, debugging packages that examine a program's internal operation), while testing may involve other tools that are discussed in portfolio 4-04-02.

In addition to dynamic testing, which involves running programs under controlled and observable circumstances, a second stage — static testing — is ordinarily included. Static testing may include manual code inspections, structured walkthroughs, or the use of automated tools that analyze software to find certain common errors not detected during the normal program production process. Contemporary testing technology advocates the combination of static and dynamic testing processes.

The input data used to cause a program to demonstrate its own features is called a test case, or test data. Individual programs can be tested (unit testing) as well as entire systems (system testing); system testing is usually divided into a set of individual tests larger than unit testing but smaller than full system testing. Testing can be done with either Black or White Box methods, depending on whether the internal operation of the program is being observed during the testing process. Normally, Black Box testing is used only at the system level; most programs are fully unit tested using White Box methods.

During White Box testing the tester observes the extent to which test cases exercise program structure. The amount of coverage indicates the likelihood of undiscovered errors. I/O relationships must be examined in detail to ensure their appropriateness.

A common strategy for testing involves establishing a minimum level of testing coverage, ordinarily expressed as a percentage of the elemental segments that are exercised during the testing process. This is called C1 coverage, where C1 denotes the minimum level of testing coverage required to ensure that each logical segment of the program has been exercised at least once during the set of tests. Unit testing usually requires at least 90 percent C1 coverage.

At the system level it is possible to define similar measures of coverage related to the extent to which subroutines are exercised. Many strategies exist for governing multiple unit testing and system level testing.

An underlying premise of this kind of structurally measured testing is that all functions in a software system must be exercised at least once. That is, it is not reasonable to accept a software system unless every part of it has been exercised. While this criterion does not guarantee discovery of all errors, it is a practical, effective method for systematically analyzing a large, complex software system.

Typical Applications

Much of contemporary testing technology was developed to meet the need to examine the quality of software embedded in products sold directly to consumers or otherwise affecting the public. The following sections examine some typical application areas and identify the primary areas of concern for software quality.

Instrumentation Systems. The use of computers, especially microcomputers, in instrumentation has increased rapidly during the last five years. For example, many oscilloscopes, data displays, signal analyzers, and communication link analyzers are more effectively built and have significantly greater capability and flexibility through the use of microcomputers. Instrument manufacturers have found, however, that they are poorly equipped to handle the software components in such instruments.

In a typical instrumentation system the software is coded at the assembly language level, a choice usually dictated by economies of program storage space and execution time. The cost of modifying such systems is extensive, particularly if modification must be performed after product introduction or if the production run is large. As a consequence, the instrument manufacturers generally provide basic levels of software testing, with the effort level based on the relative cost of repairing the software in the field.

Process Control Systems. Computers are finding increasing application in process control, particularly for those systems that have rapid real-time response requirements that cannot ordinarily be met by human controllers. Systems in this category range from nuclear reactors to automated assembly lines; the cost of failure for such control systems ranges from politically unacceptable to economically undesirable. Organizations responsible for such systems use a number of techniques to prevent failures. Triple modular redundancy is often provided for hardware in order to reduce the chance of hardware failure to a vanishingly small probability. Corresponding methods that use multiple software control processes have also been used effectively, particularly in critical applications.

Appliances. Everyone has heard the story about being "zapped" by a computer-controlled microwave oven. In fact, there is almost a competition among technologists to identify the most outrageous appliance containing a computer. (The current record is probably held by a computer-controlled wall-mounted light switch.) The implications for software in consumer products are enormous.

As with instrumentation systems, most appliance-based software is written at a very low level — sometimes even in machine code. The cost of failure is ordinarily measured as a combination of direct and indirect costs. Direct costs include repairs as well as legal protection in case of software failure. The indirect costs involve such factors as product image, organizational image, and reputation.

Automotive Computers. There has been much talk about the revolution in automotive electronics, in which many of the current mechanically implemented control functions are being replaced by full digital control. While the progress of this revolution has been slower than expected, some of today's automobiles do contain computers. In addition, several retrofit-type computers have been announced. Many of these computers are similar in function, but not in technology, to previously available "rally" computers. For this class of application, software reliability can affect the driver and passengers only indirectly. One unanswered question, however, is the extent of liability assumed by the supplier of the software/hardware combination.

Telephony. Computers are becoming the central component of telephone-switching equipment, both for the public networks and for privately maintained in-house systems. In addition to the switching function, computers are also involved in the data transfer function represented by the new value-added data transmission networks. For these applications the implications of software failure are generally less severe than those previously noted, being restricted to loss of communication or the impact of an errant call. For value-added data switching networks, however, the implications of failure are not so clear-cut. Although telephone companies have devoted substantial effort to systematic testing of computerized switching systems, and considerable experience has been gained in effectiveness evaluation of switched data networks, much quality-assurance work remains to be done.

Military Systems. Because military systems typically involve risk of human life, software quality is always emphasized. In fact, modern concepts of third-party verification and validation of software projects originated in the military community. Despite this head start, contemporary technology seems to focus on developing techniques more applicable to popular languages and computers.

Much research and development in the area of program verification has focused on military applications software.

Full verification and validation for military software is very expensive. Estimates range from $10 to $25 per statement and higher for the "full treatment." A number of companies have been active in this field for some time, with measurable effectiveness.

Liability Questions

A group of futurists recently sketched a scenario for the mid-1980s that focused on the implications for software engineering of a hypothetical multimillion dollar malpractice claim against programmers responsible for a vehicle control package. While this hypothetical application is somewhat fanciful, there are indications that liability and professional malpractice will be important issues in the future. Columnist Allen Taylor has noted the existence of "claims made" malpractice insurance available to software developers. At least one underwriter in the United States, with the backing of Lloyds of London, is now writing this kind of policy. The key question here is: What is the relationship between the cost of malpractice insurance and the kind and level of software testing being performed?

Benefits of Testing

There are additional benefits from systematic testing:

- By focusing attention on the issues of software quality, programmers as well as program testers become conscious of the need for error-free software.
- The processes involved in analyzing computer programs from the perspective of a program tester almost automatically ensure the earlier discovery of the more flagrant errors.
- Even if systematic testing does not identify significant errors, the process still serves as a backup for other quality assurance techniques, including design reviews and structured walkthroughs.
- A systematic testing activity provides a framework in which new quality-assurance technologies can be applied as soon as they become available.

ORGANIZING FOR TESTING

When planning the development of a testing function, the following questions should be answered:

- When should the program testing methodology be applied?
- How should the testing group be organized?
- How much effort should be applied to the testing activity?
- What outcomes can be expected from the testing group?

When to Test

As previously indicated, program testing requires that the computer software be relatively stable. Testing as discussed here is a post development or "acceptance test" process. A software system can be considered reasonably stable when the structure of the system is well determined (i.e., when all major subsystems and most modules are defined or at the time of first release of a working prototype).

Delaying the formal testing process until after the program is "debugged" is important because excessive changes in the software can result in extensive retesting.

Most applications of software testing technology require testing at at least the following three stages of software development:

- After individual modules have been completed and thoroughly tested by their programmers
- During integration of individual modules into subsystems or during integration of subsystems into the final system
- During final hardware/software integration

There is a weak trade-off between the onset of formal testing and ultimate quality, although this relationship is not yet thoroughly documented. Planning for program testing fairly early in the software development process is beneficial. The software engineer is thus assured of a sufficiently good set of test cases with which to initiate formal testing.

Programmers usually devise tests that exercise the working features of the program rather than identify errors. The testing process must thus concentrate on those features not previously exercised.

Organizational Schemes

How the testing team is organized, and how it fits into the software life cycle, are questions that must be addressed as early as possible. Current thought advocates beginning formal test procedures immediately following module development. The question remains, however, concerning who should perform the testing — a programmer, a separate group within the organization, or an independent outside group. Some factors to consider when choosing the organizational structure of the testing team are:

- To whom does the testing team report? If the test team reports to the ultimate buyer, its independence is not in question. If software is supplied by a subcontractor, the testing team should report to the prime contractor. In general, the testing team must have adequate freedom to deliver its report without interference.
- How is budgeting done? Testing budgets can be set as a function of total system implementation costs or as a fixed percentage of staff time, or they can be based on the size of the software system (e.g., a fixed dollar amount of testing per statement).

It is significant that a number of organizations offering quality-assurance services are springing up throughout the United States and elsewhere. Whether such organizations can survive, and how effective they will be as independent entities, remains to be seen.

THE PSYCHOLOGY OF TESTING

Testing suffers from a negative reward structure. This section discusses some of the negative attitudes that have developed concerning testing, suggesting positive attitudes that should be stressed to counter the "bad press" surrounding testing.

The Negative Side

The overall effectiveness of program testing can be influenced by prevailing attitudes. Often negative, these attitudes maintain that:

- Testing is a dirty business involving cleaning up someone else's mess without the satisfaction of creating programs.
- Testing is unimaginative; lacking the opportunity for creativity ordinarily associated with software production.
- Testing is menial work involving too many details.
- Testing is too difficult because of the complexity of programs, the difficulty of the program logic, and the deficiencies in technology.
- Testing is unimportant, and underfunded, requiring too much work in too little time.
- Testing techniques are not rigorous enough. Too much ad hoc thinking is required, and there is too little systematic knowledge to rely upon; too many special cases exist and too few generally accepted principles with demonstrated value.

- Testing has a negative reward structure; finding mistakes requires a critic's mentality.

Such attitudes can defeat an otherwise successful program testing activity.

The Positive Side

To combat these negative attitudes, it is possible to argue the following:

- Testing is a challenge that requires significant creativity and discipline to solve complicated problems.
- Testing is rewarding. Software creators appreciate detection of their oversights, managers feel more confident in the product, and program testers can take pride in the software.
- Testing is interesting, because the technology's immaturity presents opportunities for innovation New insights can be useful in developing automated tools.

Developing such attitudes can help the testing function to succeed.

SOME MANAGEMENT GUIDELINES

The following guidelines on the size, difficulty, cost, and expected outcome of a formal software testing activity are based on experience in testing software written in high-order languages (HOLs) such as FORTRAN, PL/1, or COBOL. Because these guidelines are based on estimates, they must be carefully applied They are derived from the author's experience with medium- and large-scale software systems, from statistics from the extant literature and from current work at Software Research Associates, and from reliable sources who must remain nameless. (It is fairly obvious why some organizations do not want to discuss the error content of their software!)

Size of the Testing Budget

A common way to express the size of the testing activity is as a percentage of total software development cost. For HOL-coded systems, experience suggests that budgets in the range of 25 to 45 percent of total development cost are appropriate. Here, total development cost includes all expenditures up to the first release of the software. The amount of effort devoted to formal testing is a function of how critical the system is. The higher percentage figure appears to be minimally adequate for the highest-priority software system. Examples of systems in this class are those that are man-rated (where software failure may endanger human life) and those that have a sizable production run (with value far exceeding the total software development cost).

A second rule of thumb in determining testing budgets is to provide "insurance" based on the expected cost of failure. For a product with an embedded computer, for example, it would be reasonable to allocate 5 to 10 percent of the total cost to repair the first software failure (in the field if necessary) to systematic testing. Developers should assume that there are errors in the software: the only question is whether the errors will be discovered before or after the product is issued. Such an attitude allows a product to be released with a known error if the cost to repair it at the time is too extensive (e.g., if the program is already "burned" into ROM), or when the impact of the problem is judged too small to warrant changes before the next model is issued.

Size of the Testing Problem

Past experience in software analysis is useful in estimating the total difficulty of the testing activity for HOL-coded software systems. These estimates vary widely depending on both the type of problem being solved and the style of programming.

To understand these estimates it is important to define some testing terms:

- A *statement* in a program is either a declaration or an action statement and may run to more than one line of text.
- A *segment* is a logically independent piece of a program that corresponds to the actions the program takes as the consequence of a program decision (including the decision to invoke a subroutine). Thus, a program with no logic has one segment, each IF statement contributes two segments, and so on.
- A *test* is one transfer of control to the software system (regardless of the part being examined) and the resulting actions taken by the system. Thus, a test in effect corresponds to a single subroutine invocation.

A subroutine with no logical statements (a so-called straight-line program) has one segment consisting of all statements of the program: at least one test is required. Coverage is measured (in the C1 measure) by the percentage of segments that are exercised at least once over all tests run.

In typical programs the number of segments equals between 5 and 25 percent of the number of statements (i.e., each segment typically corresponds to about 4 to 20 statements). Multiplying the number of statements by 10 percent provides a useful initial approximation of the number of segments.

The number of tests required to achieve 90 percent coverage is also fairly constant. For typical HOL-coded programs, each test exercises an average of 25 to 35 statements. This figure should not be misinterpreted; achieving 90 percent segment coverage may involve some tests that exercise only a few statements as well as others that exercise as much as 50 percent of the program.

The figure of 90 percent C1 (segment level) coverage was chosen because it represents a reasonably attainable goal for a large software system. Bringing the figure to 100 percent, if at all possible, would in most cases substantially increase the number of tests required.

The Results of Testing

The most important question is: Given 90 percent coverage of segments in a complex HOL-coded program, what is the likelihood of finding errors of varying severity? Since the number of errors found is a function of the skill of the program testing team, the quality of the software, and the amount of prior testing performed, it is difficult to answer this question precisely.

For a typical situation, in which software has been fully unit tested by the programmers but not systematically tested by a separate group, testers can expect to find errors in 0.5 to 2 percent of the number of statements. This suggests that a 1,000-statement program, when completed by the program development staff, may contain as many as 20 errors of varying severity. Establishing two severity levels makes it possible to refine these estimates. If errors are classified as either "serious" or "fatal," only about 15 percent of the deficiencies will be "fatal."

RECOMMENDED COURSE OF ACTION

This portfolio has presented an overview of software testing to help the DP manager to:

- Understand the basics of testing and the expected benefits
- Organize a testing function
- Foster positive attitudes about testing
- Determine the extent of testing needed
- Develop a realistic budget for testing

The following two portfolios in this series discuss testing tools and methodologies to help managers choose the most effective techniques for their organizations.

This portfolio was written by Edward F. Miller, Software Research Associates, San Francisco CA.

NOTES

INSIDE:
Description of Tools, Static Testing Tools, Dynamic Testing Tools, Future Developments

Software Testing Tools

PAYOFF IDEA. Quality assurance testing is becoming an increasingly important part of the software development effort. Because this area is heavily labor intensive, automated tools for testing software have been extensively researched. Although much remains to be done, many tools already have been used effectively and efficiently to improve software quality. This portfolio provides an overview of the kinds of tools now available and discusses their use.

PROBLEMS ADDRESSED

Because of the highly complex nature of the analyses involved, software testing efforts are virtually dependent on the use of automated tools. This portfolio describes the general types of software testing tools, explains their functional characteristics, and suggests ways in which these functions can be integrated into a single testing system for use on complex software.

DESCRIPTION OF TOOLS

There are two broad categories of software testing tools: static analysis tools and dynamic analysis tools. Most tool functions fall into one category or the other; however, there are exceptions such as symbolic evaluation systems and mutation analysis systems (which actually run interpretively). The following are the main software testing tools and their functions:

- Static analyzers — examine programs systematically and automatically
- Code inspectors — inspect programs automatically to ensure that they adhere to minimum quality standards
- Standards enforcers — impose simple rules on the programmer
- Coverage analyzers — measure the value of Cx, where x = ?
- Output comparators — determine whether the output in a program is appropriate
- Test file/data generators — set-up test inputs

- Test harness systems — simplify test operations
- Test archiving systems — provide program documentation

Each of these tools is discussed in the sections below. The discussion is organized according to the two major types of tools — static and dynamic. The tools mentioned are described in more detail in reference 3.

STATIC TESTING TOOLS

Static testing tools analyze programs without executing them. Examples of static testing tools are discussed in the following four sections.

Static Analyzer

A static analyzer uses a precomputed data base of descriptive information derived from the source text of the program. The purpose of a static analyzer is to "prove" allegations (claims about the analyzed programs) by systematic examination of all the cases.

These tools are very similar to code inspectors (see the following section), but static analyzers are stronger. Examples of static analyzers are FACES, DAVE, RXVP, and PL/I Checkout [3]. These analyzers are all language and/or system dependent (i.e., they can be used only with a particular language or system).

Experience. For many applications, static analyzers report 0.1 to 2.0 percent deficiency. Some of these deficiencies are real; however, some of these reports are false warnings that are ignored after interpretation

Possibility. Build a static analyzer for the language/system combination being used; then apply it to all code produced for this system.

Assessment. Can produce a favorable cost/benefit ratio if the software analyzed is critical and the tool is well designed.

Problems. Dependence on language vagaries, variance between compilers, high initial costs.

Code Inspector

A code inspector does the simple job of uniformly enforcing standards for many programs. These standards can be single- or multiple-statement rules. It is also possible to build code inspector assistance programs that can be used interactively to "force" a human inspector to do a good job.

The AUDIT system is a typical example of a code inspector; it enforces some standards and imposes some minimum conditions on programs. In use by the Navy for some time, AUDIT has been found effective in production coding [3]. Code inspection activity is found in some COBOL tools (e.g., ADR's Librarian system), and in some parts of tools such as RXVP.

Experience. Found useful in many circumstances.

Possibility. Implement an automated code inspector system for use in handling production coding and possibly non-production coding.

Assessment. Used properly, this tool can have a big payoff.

Problems. Language dependence, programmer resistance, initial investment costs, difficulty in constructing a good programmer interface.

Standards Enforcer

This tool is similar to a code inspector, except that the rules used are generally simpler. The main distinction between the two types of tool is that a full-blown static analyzer examines whole programs, while a standards enforcer examines single statements.

Since only single statements are treated, the kinds of standards enforced are often cosmetic. Even so, these standards are valuable since they enhance the readability of the programs. The readability of a program is an indirect indicator of its quality.

Experience. After initial programmer resistance, these tools are often helpful in filtering out completely unreadable code.

Possibility. Establish standards and support tools to enforce them and then require all outside suppliers of software to meet those standards.

Assessment. Indirect beneficial effect on software quality.

Problems. Programmer resistance, arguments that standards concerning program format are not important

Other Tools

Related tools are used in software testing to catch bugs indirectly through program listings that highlight mistakes. One such tool is a program generator that is used (mainly in COBOL environments) to produce the pro forma parts of each source module. Use of such a method ensures that all programs "look alike"; this in itself enhances their readability. Another tool is a structured programming preprocessor that produces "pretty print" output. The augmented program listings produced by this tool usually have automatic indentation, indexing features, and so on (reference 3 mentions a number of such tools).

DYNAMIC TESTING TOOLS

Dynamic testing tools support the dynamic testing process in a variety of ways. Most of the main functions are covered in the following sections. In addition to individual tools that accomplish these kinds of functions, there are a few integrated testing systems that group the functions together.

A test is defined as a single invocation of the test object and all the execution that ensues until the test object returns control to the point

28

where the invocation was made. Subsidiary modules called by the test object can either be real or can be simulated by testing stubs. A series of such tests is normally required to fully test one module or set of modules. Test support tools must perform these functions:

- Input-setting — selecting the test data that the test object reads when called
- Stub processing — handling outputs and selecting inputs when a stub is called
- Results displays — providing the tester with the values that the test object produces so that they can be validated
- Test coverage measurement — determining the effectiveness of the test in terms of the program structure
- Test planning — helping the tester to plan tests that are both efficient and effective in discovering defects

All of these ideas are discussed in the following sections describing the types of dynamic testing tools.

Coverage Analyzer (Execution Verifier)

Coverage analyzers or execution verifiers (also called automated testing analyzers, automated verification systems, etc.) are the most common and the most important testing tools. They are often relatively simple.

C1 is the most commonly used measure of testing coverage. Most often, C1 is measured by planting subroutine calls — called software probes — along each segment of the program. The test object is then executed, and some type of runtime system collects the data, which is then reported to the user in fixed-format reports. A practical minimum value for C1 coverage is 85 to 90 percent.

Experience. Coverage analysis can be used in most quality assurance situations, although it is more difficult when there is too little space or when non-real-time operation is not equivalent to real-time operation.

Possibility. Require C1 coverage (at least) in all cases.

Assessment. Highest payoff in terms of quality at the lowest possible cost using C1 measurement.

Problems. Programmer resistance, nonstandard system use, difficulty in interpreting C1 values for multiple tests (such interpretation requires some type of history analyzer).

Output Comparator

Output comparators are used in dynamic testing — both single- and multiple-module (system level) testing — to check that predicted and actual outputs are equivalent. This is also done during regression testing (testing during the system maintenance phase). The objective of an output comparator system is to identify differences between two files, one the old and the other the new output from a program. Operating systems

for some minicomputers often have an output comparator (sometimes called a file comparator) built in.

Experience. Finds differences effectively and usually quite efficiently.

Possibility. Equip the software testing facility with an output comparator for general use.

Assessment. Needed tool for a quality assurance environment.

Problems. May produce too much output if the old and new files are lengthy or have many differences.

Test File Generator

A test file generator creates a file of information that is used as input to the program. The file is created based on commands given by the user and/or data descriptions (e.g., in the data definition section of a COBOL program). This is primarily a COBOL-oriented testing concept in which the file of test data is intended to simulate transaction inputs in a data base management environment. The concept can, however, be adapted to other environments.

Experience. Produces benefits when this type of tool is called for.

Possibility. Use test file generator, or adapt the concept, to create lengthy files of input transactions. For the most effective use, have the input data varied automatically to account for a range of cases.

Assessment. A test file generator system — either commercial or "home brew" — can save testing time and effort in some situations.

Problems. Primarily COBOL-oriented.

Test Data Generators

Test data generation is a difficult problem for which no general solution exists [2]. There is, however, a practical need for methods to generate test data that meets a particular objective (normally to execute a previously unexercised segment in the program). Practical approaches to generating automatic test data are limited by technical difficulties [2].

In practice, variational test data generation is often quite effective. In this technique the test data is derived (rather than created) from an existing path that approximates the segment for which test data is to be found. This is often very easy to do, apparently because the program's structures tend to assist in the process.

Experience. Automatically generating test data is practically impossible, but research is now being done on the problem.

Possibility. Use variational and machine-assisted methods only.

Assessment. Technical issues of this problem may be more formidable than the problem is in practice.

Problems. Technical difficulties of the test data generation problem: difficulty in developing good heuristics for interactive test data generation.

Test Harness Systems

A test harness system is link-edited and bound around the test object. This type of system permits easy modification and control of test inputs and outputs and provides for on-line measurement of C1 coverage values. Some test harnesses are batch oriented, but the high degree of interaction available in fully-interactive test harness systems makes them very attractive for practical use [1].

Today's thinking favors interactive test harness systems, which often become the focal point for installing many other types of analysis support.

Experience. Clear cost/benefit improvement, even with batch oriented systems. (TESTMANAGER, a typical example of a batch system, has nearly 300 installations.)

Possibility. Current technology enables development of a test harness system for almost any system or language.

Assessment. Give strong consideration to building a test harness system for both single-module and system testing.

Problems. Requires an interactive environment. Must be customized for the particular language/machine combination. Development cost.

Test Archiving Systems

The purpose of a test archiving system is to keep track of a series of tests and to document that the tests have been performed and that no defects were found. A typical design requires establishing procedures for handling files of test information and for documenting which tests were run, when they were run, and with what effect.

Experience. Test archiving systems are mainly developed on a system- or application-specific basis.

Possibility. Implement manual methods at least, and give consideration to automating test documentation processes.

Assessment. High potential value during regression testing (during the system maintenance phase).

Problems. Dependence on local environment, inadequate experience with automated documentation methods.

FUTURE DEVELOPMENTS

The future development of tools will depend on two major factors:

- The level of use of current tools (both individual tools and integrated sets of tools)
- The degree of interest expressed by the software engineering community in having advanced tools

The development of tools tends to precede the application of methodologies (i.e., when a new testing methodology is conceived it is normally necessary to construct, or at least envision, a set of tools to implement it). This means, in effect, that each new testing methodology, invented by researchers must be tried out first with prototypes of what may ultimately become integrated tool sets.

This occurred in the early 1970s with such systems as JAVS and RXVP; early experimental prototypes were developed in most cases without complete information about their ultimate operational function. If this pattern is translated into a projected trend, it can be predicted that in 5 to 10 years the third or fourth generation of software testing tools will be relatively widely used.

In the future, software testing tools will be:

- Interactive support facilities
- Connected to data base systems used to keep track of detailed information
- Effectively built in to the particular system and/or testing methodology

Portfolio 4-04-03 will continue the discussion of software testing by examining levels of testing and different testing methodologies.

This portfolio was written by Dr. E.F. Miller, Software Research Associates, San Francisco CA.

References

1. Miller, E.F. "Some Statistics from the Software Test Factory," *Software Engineering Notes* (ACM/SIGSOFT), January 1979.
2. Miller, E.F. and Howden, W.E. *Software Testing and Validation Techniques*. Long Beach CA: IEEE Computer Society, 1979.
3. Software Research Associates. *Automated Tools for Software Engineering, Tool Index*. SRA TN-674, February 15, 1980.

INSIDE:
Levels of Testing, Inspection Methods, Testing Methods, Sample Applications

Software Testing Methods

PAYOFF IDEA. An effective software testing program can significantly improve the quality and maintainability of software systems. There are a variety of testing methods and several levels of testing that can be used. Some of these methods are already in practical use; however, others are still in the experimental stages. To help the DP manager develop a testing program, this portfolio discusses the various methods, their uses, and the results that can be expected.

LEVELS OF UNIT TESTING

Software quality assurance testing consists of a set of disciplines that can be applied in a number of ways and with varying levels of sophistication. There are three essential steps in a typical quality assurance activity:

1. Systematic inspection and analysis of code, seeking discrepancies between stated requirements and the actual software
2. Dynamic-test planning, defining a series of tests that demonstrate various functions of the software
3. Comprehensive test evaluation, both during the dynamic testing and after tests are run

These three steps are included in one form or another in almost all testing processes, although not necessarily under precisely these names [1]. The code inspection process is fundamentally different from the other two, which involve execution of the program. Thus, the technology for systematic inspection of programs can be handled in a completely different way than dynamic testing methods.

This portfolio describes software testing methods in order of increasing sophistication, expressed in terms of coverage measures that indicate how thoroughly the processes in steps 2 and 3 are actually accomplished. The term "coverage measure" is used here as a basis for describing the various methodologies. The measures are structural indicators of the thoroughness with which each portion of the software is actually exercised. The relationship between the coverage achieved and the chance of an error remaining in the software is indirect at best, even though there is substantial evidence to indicate that the correlation between high coverage and fewer errors is very high [2]. Definitions of the various coverage measures are given at the beginning of each section.

INSPECTION METHODS

The concept of "code inspection" was first described in full by Fagan, who described IBM's internally developed procedures [11]. These procedures are an outgrowth of structured programming and can be easily combined with regular structured programming methods. When applied to critical software, code inspection can be expected to uncover several types of problems. According to Fagan, code inspection is best performed by teams in which individuals have very well-defined roles:

- The moderator responsible for preserving the objectivity and integrity of the inspection effort
- The designer of the system
- The coder/implementor who built the software
- The tester responsible for all testing of the software

This team applies a set of well-defined rules to the candidate software and looks for errors in the programs. The errors are either corrected immediately or held for later verification through dynamic testing.

> *Experience.* Fagan's report, which seems to be the standard for this technique, claims error detection rates of 80 percent or more, at costs that are quite low: between 650 and 900 noncommentary source statements (NCSS) per team hour for preparation and inspection at two stages. (This approximates 200 NCSS per hour overall.)
>
> *Possibility.* Code inspection generally increases the overall productivity of the final product by reducing error content early.
>
> *Assessment.* Use code inspection methods as software is delivered as part of a software quality assurance program.
>
> *Problems.* Rules to be followed are specific to each language/machine/application combination, so they must be redone for each new system.

TESTING METHODS

Testing methods always focus on the execution characteristics of the programs being examined. A main trend in the testing literature is to interpret the effect of a test in terms of the level of structural exercise the programs attain [2]. The levels then have a specific meaning in terms of testing coverage. The next section describes the various testing levels.

The testing process assumes a constant program structure viewed through a model called a "directed graph representation" of the program. In this model, nodes correspond to "places" in the program text, and edges correspond to actions (segments).

Ad Hoc Testing (C0 Coverage)

Conventional programming methods normally involve only a limited amount of testing — sometimes called "debugging" — and typically result in less than full exercise of a program. Many writers during the 1970s advocated the C0 measure as a way to characterize the quality of a set of tests. C0 is the percentage of the total number of statements in a module that are exercised, divided by the total number of statements present in the module.

The intention of this measure is to ensure that the highest possible fraction of program statements are exercised at least once. Unfortunately, programmers often test programs very naively. C0 is not normally considered an acceptable level of testing in the quality assurance process; however, it is generally felt to be better than nothing at all [22].

> *Experience.* Most programs, when completed, are 75 to 90 percent tested in the C0 measure but only approximately 50 percent tested in the C1 measure (see the next section).
>
> *Possibility.* Use C0 testing as the basic measure of programmer testing, but do *not* use it to measure actual quality assurance testing coverage.
>
> *Assessment.* Use C0 only if no other measure is possible.
>
> *Problems.* Even if achieved, C0 leaves some segments unexercised

Basic Testing (C1 Coverage)

The next level in testing technology is the first truly structure-based measure, C1, which is defined as the percentage of segments exercised in a test as compared with the total number of segments in a program.

The C1 measure was developed through research aimed at finding a strong measure for testing effectiveness that takes into account most, if not all, of the elementary features of a software system [2]. C1 measures the total number of segments exercised in each test or, when computed cumulatively, the total fraction of segments that are exercised in one or more tests in a series. The goal of C1 is to have a set of tests that, in aggregate, exercise a high percentage of all of the segments that can be exercised.

Current thinking holds that a C1 value of 85 to 90 percent is a practical level of testing coverage. Experience suggests that this level of C1 coverage is adequate to discover perhaps 90 percent of the available errors, although there is no way of knowing for certain. The lower limit of 85 percent was established in 1973-74 when the JAVS system was purchased by the U.S. Air Force with an 85 percent self-test requirement.

Although 85 to 90 percent coverage is not perfect, it is much stronger than the actual 25 to 50 percent achieved by most programmers. Even when coverage information is provided as part of the programming process, evidence suggests that unless prodded, programmers do not exceed the threshold levels for C1 coverage.

Experience. Attaining 85 to 90 percent C1 coverage usually finds 2 to 5 percent NCSS defects, approximately 10 to 20 percent of which are probably fatal or dangerous.

Possibility. Apply C1-based measurement as soon as possible on critical applications.

Assessment. C1 is a good starting point for a systematic methodology for testing, and it is a good minimum requirement.

Problems. No proof exists to back up the empirical evidence that software testing using the C1 measure has a predictable effect in forcing discovery of errors.

Intermediate Testing (C1+ Coverage)

The next level above C1 testing attempts to exercise some basic features of the program in addition to testing all of the segments. This is the C1+ measure, which assesses the quality of tests by requiring full C1 coverage, plus one test for the interior and exterior of each iteration.

This measure was developed to emulate proof of correctness methods [2,14]. The C1+ measure normally requires more tests than C1 alone. Although there is only a small amount of evidence, it is felt that in practice C1+ is significantly stronger than C1, because C1+ is more likely to uncover looping errors in programs, in much the same way that a proof of correctness discovers such errors.

Experience. There has been minimal experience with C1+, but serious consideration should be given to using this measure.

Possibility. Apply C1+ in the most important third of the application programs.

Assessment. C1+ is a likely candidate for future use.

Problems. C1+ is often structure dependent, so if a program has a structural fault there will be less chance of finding defects (this is a common limitation of all structure-based measures).

Advanced Testing (Ct Coverage)

Ct is a very strong testing level, for which tests are designed in terms of the pure-structured representation of a program. The definition of Ct is: the percentage of independent execution subtrees in the hierarchical decomposition tree of a program that have been exercised, in terms of all of the possible subtrees that can be found for that program.

For a pure-structured program, Ct involves the same kind of testing that would be accomplished in a full-blown proof of the program. In other words, the set of tests used in this method is the same as the set that would be used if the "verification conditions" [12] for the program were tested.

The Ct measure is computed as follows: the hierarchical decomposition tree is generated from the directed graph (digraph) of the program, and the set of all possible subtrees of this tree is found. Each subtree represents one distinct executional class for the program. Ct is measured as the percentage of all such subtrees that have been tried during testing. Another way to find the subtrees is to think of the program as if it were purely structured; then choose the tests by inspecting that representation of the program. Decomposition methods are not necessary when the program is already pure-structured.

Experience. Ct has been used primarily in pencil and paper investigations although some organizations have automated test planning methods based on this principle.

Possibility. Use Ct as the basis for very strong testing of modules of applications software.

Assessment. Ct will probably be the strongest of the dynamic testing disciplines.

Problems. Ct is highly dependent on the program structure and may not be effective when there are structural faults.

Symbolic Evaluation Methods

Aside from proof of correctness methods, symbolic evaluation/execution is the strongest method known for ensuring quality through testing. Symbolic-evaluation is a static analysis method because the program is never actually executed. Instead, a particular path through the program is evaluated in detail from the source-level version of the program. All of the indicated computations are performed symbolically, subject to constraints that may exist along the path because of the kind and number of conditionals that specify whether the path is executable. The result is a set of formulas that tell what the path "computes."

This method is very similar to proof of correctness [12] and together with test data generation systems and proof systems, is considered the basis for future quality assurance systems.

Experience. Symbolic evaluation methods are used primarily in academic environments and for small (less than 2,000 NCSS) programs. Many errors are found during the symbolic evaluation process; however, the method appears to be effective only when used as an interactive testing tool.

Possibility. Further research could lead to a user-interactive system that reduces the amount of work to be performed.

Assessment. There are good prospects for this approach in future testing efforts.

Problems. There are difficulties with combinatoric growth in the formula size, with complex formulas and path predicates, and with reducing the formulas to human-readable format.

Comparing Testing and Program Proving Methods

A proof of correctness is a mathematical demonstration of consistency between a program and its specifications, in relation to sets of statements about the environment and the semantic behavior of the program. The largest program ever proved correct consisted of approximately 1,700 statements, and that proof was done as part of a university-level research project [14]. Forty-three errors were found; however, it was later discovered that about 86 percent of these errors would have been detected through C1 testing.

In the case of some proofs that have been completed and published in the technical literature, the programs have later been found to contain easily detectable errors. This indicates the fundamental dependence of proof methods on mathematically complete statements about the environment in which the programs are supposed to run [13].

The basic steps in a proof are:

1. Establishing the assertions
2. Constructing the proof
3. Checking the result

These steps are only partially automatable. Finding the assertions is a process similar to that involved in setting up the plans for C1-based testing. The proof can be constructed automatically; however, most systems require human assistance when given a real-world problem to work on. In addition, the check process itself must be verified by running actual tests of the program.

Experience. Much of the experience has been in the research and development community, little in the practical world.

Possibility. Scaling problems will limit proving methods to smaller systems for several years.

Assessment. Use proofs for most important modules only.

Problems. There is difficulty in checking the proof. Known errors in proof of correctness exercises as well as the complexity of the method are further problems.

Future Techniques

The preceding sections have discussed the basic methods of software testing in terms of the kind of coverage that must be achieved in each level for single-module analysis. The methodology for multiple modules and for systems is similar; the differences result from problems of complexity and size. Future methods of testing will include domain testing and mutation analysis [2].

Domain Testing. Domain testing [16] involves analyzing the input space of each module to establish a set of domains that can be used as the basis for systematic demonstration of the module's behavior. The input domain boundaries are analyzed in detail to determine their intersections; test data that pushes the behavior of the program close to the boundaries is deemed more useful than ad hoc test data.

Mutation Analysis. Mutation analysis involves systematically constructing small variations (mutants) of the given program and then determining if they can be distinguished by the existing test data. Each mutant that is distinguished is termed "retired"; the remaining mutants are either equivalent to the original program or inequivalent but undistinguished by the test data. In the latter case, test data must be added [15]. Although mutation analysis is still experimental, it is the focus of a great deal of interest because it provides a direct, quantitative measure of the quality of test data for a program. The disadvantage of the method is that it may require rather large numbers of mutants, and the number of equivalent programs may be quite large.

Experience. Although the method shows great promise, it has been used only in a research environment.

Possibility. Such methods as domain testing and mutation analysis will be used by the late 1980s as the basis of quality assurance testing.

Assessment. Both methods are too immature for real-world application.

Problems. Domain testing appears to be limited in effectiveness to linear-constraint (linear logic) applications. Mutation analysis is still in the experimental stages and needs extensive field validation prior to use in a real-life situation.

SAMPLE APPLICATIONS

This section describes some of the basics of two major classes of testing: single-module testing and multiple-module (or system) testing.

(These two classes are discussed briefly in portfolio 4-04-01.) The next paragraphs discuss the use of these two methods in testing four cases involving different types of software systems. The four cases are:

Case A: A high-criticality single module
Case B: A medium-sized, medium-criticality system
Case C: A large, medium-criticality system
Case D: A large, high-criticality system

Case A is in a high-level language, is 250 to 500 statements long, and has a very complicated control structure. The module has many different functions and must be error free.

Case B is a set of 25 modules that supports an on-line facility. Software failure does not result in serious loss because there is an automated backup system; however, it is expensive in terms of lost production and associated waste. The system is written in a structured extension of FORTRAN and contains approximately 6,500 statements. The calling depth in the structure of the system is between four and seven (it is neither "flat" nor excessively complex).

Case C is a comprehensive system for control of a facility; 80 percent of the system is in a high-level language like PASCAL or Ada, and 20 percent is in an assembly language. The total volume of code is in the range of 175,000 NCSS. If this system fails, there is a substantial loss of value; however, no lives would be lost and the damage would not normally be extensive and/or expensive to repair.

Case D is a geographically dispersed interactive control system used where human life is at stake (e.g., a very advanced air traffic control system). The system approaches the limits of current complexity in that the latest methods are employed in its design and implementation. The total volume of code is approximately one million NCSS.

Single-Module Test Methodology

Single-module testing puts maximum focus on each separately compilable and/or separately invokable software module. The exhaustive analysis of a single module has three stages.

First is an analysis and preprocessing stage during which the module is structurally analyzed and studied. If an automated test system is used, this stage includes preparing the module to be the test object in the subsequent phases. Typically some programmer-supplied test data that can be used to find an initial value for the testing coverage measure is available. The coverage level obtained by these initial tests is normally quite low. In fact, poor testing by programmers is a major reason for high defect rates in software [2].

In the second stage, tests are generated systematically to increase coverage. Usually, only a few errors are found; although the programmer may not have exercised the program thoroughly, many errors that can be found through the inspection process have already been removed.

During the final stage, it may be necessary to plan for a number of very difficult tests so that all the segments can be exercised. As an alternative to exercising a segment, the program tester can spell out the reasons why it was not exercised.

Experience. Most errors are found near the end of the single-module testing process, although the reasons why this is true are not clear [9].

Assessment. Every module should be tested to at least a high C1 level and much more testing should be done if possible.

Problems. Single-module testing is much easier with the right tool environment (i.e., an interactive test system); without these tools the work is very difficult.

System Testing Methodology

System testing is analogous to single-module testing, except that a segment corresponds to a whole module. The two modes of testing can be very different, depending on the application software and its internal structure.

System testing is done either bottom-up or top-down. If the bottom-up methodology is chosen, the flow of attention during the testing process is from single modules at the "bottom" of a system to the topmost (driver) modules later on. If the top-down method is used, attention begins at the topmost module. Stubs are sometimes used to simulate subsidiary modules.

The choice of one of these methods depends on many factors, but primarily on the structure of the software system itself. Large systems that are "flat" (those that have a small maximum invocation chain length) usually work better in bottom-up mode. Small systems that are very deep usually fall into the top-down mode.

Experience. Most experience is with single-module testing; however, in some cases there is extensive system-level experience. Early data suggests that there is only a quantitative difference between the two levels of testing.

Possibility. System testing is a direct extension of single-module testing to the system level, provided the right input/output data is available.

Assessment. Proceed with system testing as if it were module testing.

Problems. There are problems with lack of experience in testing very large systems and unavailability of appropriate tools.

Application to Cases

These methodologies should be applied to the four cases listed previously as follows

- Case A — high-criticality single module. In this case, the strongest methodology available for single-module testing should be applied to the critical module. The Ct measure should be used if possible.

- Case B — medium-sized, medium-criticality system. Here it is not possible to focus all of the effort on each module; therefore, a limited approach to maximize the payoff from rigorous single-module testing should be followed. This maximizes the number of errors found.
- Case C — large, medium-criticality system. Large systems usually have a somewhat larger budget; therefore it is possible to consider building special-purpose tools to support the testing activity. This level of effort can involve other quality assurance methods in addition to simple testing.
- Case D — large, high-criticality system. In this situation the best methodologies must be used to ensure thorough analysis of single modules throughout the software and an effective level of system testing.

Contemporary Results

It is possible to set up a systematic test "factory" or assembly line [20]. Software Research Associates created an experimental automated software test factory with the goal of processing a large amount of software at relatively low cost. In 24 man-months of work in this test factory, 60,881 statements (4,378 segments) of a PL/1-like language were processed. A total of 1,544 tests were required to achieve 89.7 percent overall C1 coverage.

In this processing, 1,486 discrepancies were found between the software and the specifications. Approximately 190 of these were fatal program errors; the remainder were deficiencies that were accepted as such by the client organization. The main lesson learned in this case was the necessity for a good tool environment, the costs of which are not reflected in the preceding figures.

In another activity, single-module and system-level testing were performed on air traffic control support software [10]. A total of 23,700 statements in JOVIAL/J2 were tested, and 98 percent C1 coverage was obtained. The results of the error-detection process were as follows:

- 3.57 percent NCSS defect rate in unit testing
- 0.25 percent NCSS defect rate in subsystem testing
- 0.08 percent NCSS defect rate in system testing
- 3.91 percent overall defect rate

The importance of these values is that they were found by systematic examination of an existing software system.

A final example comes from the nuclear energy field [21]. In this experiment, C1-based testing was performed on programs to determine whether multiple programming teams using different programming languages could produce significantly better software.

Using a coverage-analysis system based on RXVP the team was able to discover 100 errors in approximately 4,500 statements, a 2.4 percent error rate. Most of these errors were in the specification — the programs had been written correctly even when the specification was faulty.

Experience. When using good tools in a positive atmosphere, a testing effort can find a significant number of errors, defects, and deficiencies in completed programs.

Possibility. Continued emphasis on this technique can reduce the errors appearing in programs.

Assessment. Costs for full C1-based testing add approximately 20 to 40 percent to programming costs (private sector cost estimate).

Problems. Testing effort requires good tools and strong management support for quality assurance. A fully comprehensive methodology is still unavailable (C1 does not guarantee the discovery of errors). The high cost of a suitable tool environment is also a problem.

Future Results

Predicting future trends is difficult in a technical area as volatile as software testing. Likely future trends can be divided into three categories: theory (which tends to lead to methodological advances), supporting tools, and methodologies.

Theory. This is the most difficult area for making predictions. It is reasonable to expect that most of the technical issues surrounding software testing will be largely resolved by the mid-1980s; however, there is no guarantee of this. Unanswered questions that must be dealt with include:

- What is the direct relationship between coverage analysis and testing?
- How can testers know when errors can no longer be present in a program? (Current testing procedures can only verify that a program is consistent with its specifications; they cannot establish the program's validity in "real-world" application.)

Tools. Contemporary tools, such as those described in the preceding sections, are still far from ready for full production; they are still prototypes being evaluated for eventual use. Each new large-scale project brings greater understanding of how tools should operate; however, there is still much to learn.

Certain features that will be necessary in future tools can be discerned from current experience:

- Interaction with the program tester is very important because it helps him or her to pay attention to more detail
- Documentation backup systems are critical in tools to ensure that enough data is preserved.
- Structural test planning methods are crucial for eliminating the burdens of test-planning computations.

STANDARDS, PRACTICES, AND DOCUMENTATION

Methodologies. Theoretical developments usually precede the development of formal testing methodologies. The current art of testing is based primarily on systematic structural and behavioral analysis of software, and this basis can be expected to remain for quite some time.

New standards for quality assurance may have a positive effect on the kinds of methodologies actually practiced. For example, if minimum level-of-coverage criteria are adopted for acceptance testing of software systems, quick adoption of corresponding systematic testing methodologies can be expected. In some areas, standards for software quality have already been created, although not without some controversy [9].

Methods for system-level testing are less well developed than single-module methods, and the greatest improvements can be made at the system level. At present, more experience is needed in order to produce better methodologies.

This portfolio was written by Dr. E F Miller, Software Research Associates, San Francisco CA

References

1 Alford, M W "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Transactions on Software Engineering*, January 1977.

2 Belford, P C, Berg, R A, and Hannan, T L, "Central Flow Control Software Development: A Case Study of the Effectiveness of Software Engineering Techniques," *Proceedings of the 1979 International Conference on Software Engineering*, Munich, 1979

3. Brooks, F P *The Mythical Man Month: Essays on Software Engineering*. Reading, MA: Addison Wesley, 1975

4 Cooper, J D, and Fisher, M J *Software Quality Management*. New York: Petrocelli Books, 1979

5 DeMillo, R A, Lipton, R J, and Sayward, F G, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, Vol 11, No 4 (April 1978)

6 Fagan, M E, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol 15, No 3 (1976)

7 Gerhart, S L, and Yelowitz, L, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Transactions on Software Engineering*, September 1976

8 Glass, R L *Software Reliability Guidebook*. Englewood Cliffs NJ: Prentice-Hall, 1979

9 Hantler, S L, and King, J C, "An Introduction to Proving the Correctness of Programs," *Computing Surveys*, Vol 8, No 3 (September 1976)

10 Miller, E F, "Automated Tools for Software Engineering," Catalog No EHO 150-3, IEEE Computer Society, October 1979

11 Miller, E F, "Program Testing Technology in the 1980s," *The Oregon Report: Proceedings of the Conference on Computing in the 1980s*, IEEE Computer Society, 1979

12 Miller, E F, "Some Statistics from the Software Test Factory," *Software Engineering Notes* (ACM/SIGSOFT), January 1979

13 Miller, E F, "Validation of Requirements Engineering: Final Report," Software Research Associates, SRA TR 640, June 1979

14 Miller, E F, and Howden, W E *Software Testing and Validation Techniques*. Long Beach CA: IEEE Computer Society, 1979

15 Myers, G J *The Art of Software Testing*. New York: John Wiley & Sons, 1979

16 Myers, G J *Software Reliability*. New York: John Wiley & Sons, 1977

17 Patterson, D L, PhD Thesis, University of California, Los Angeles, 1978

18 Ross, D T, and Shoman, K E, "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, January 1977.

19 Software Research Associates, "Automated Tools for Software Engineering: Tool Index," SRA TN 674, February 15, 1980

20 *Software Testing: The State of the Art*. Maidenhead, England: Infotech International Ltd, 1978

21. Special Issue on Program Testing, *Computer*, Vol 12, No 8 (April 1978).

22. Special Issue on Software Quality Assurance, *Computer*, Vol 13, No 4 (August 1979).

23. Stevens, R T *Operational Test and Evaluation: A Systems Engineering Process*. New York: John Wiley & Sons, 1979

24 Teichroew, D, and Hershey, E A, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, January 1977

25 Trauboth, H, et al, "Program Testing Techniques for Nuclear Reactor Protection Systems," *Software Testing: The State of the Art*. Maidenhead, England: Infotech International Ltd, 1978

26 Wegner, P *Programming with Ada: An Introduction by Means of Graduated Examples*. Englewood Cliffs NJ: Prentice-Hall, 1980

27 White, L J, and Cohen, E I, "A Domain Strategy for Computer Program Testing," *Proceedings of the IEEE/NBS Workshop on Software Testing and Test Documentation*, Ft Lauderdale FL, December 1978

28. Yourdon, E *Structured Walkthroughs*. New York: Yourdon Press, 1979

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

LIBERACION DEL SISTEMA
DOCUMENTACION COMPLEMENTARIA

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

ix. LIBERACION DEL SISTEMA
(DOCUMENTACION COMPLEMENTARIA)

6. RELEASE PHASE

TABLE OF CONTENTS

The purpose of the *RELEASE PHASE* is to establish that the system is acceptable for normal use. Upon completion of the Release Phase. the development project will be closed and the system will be maintained by ongoing support activities. The Release Phase sign-off signifies that *management believes* the system is acceptable, supportable, and distributable to the intended users. It also means the development cycle is complete.

Two sets of activities take place during the Release Phase:

o implementation of the Support Plan that began in the previous phase

o beta testing of the system to prove it acceptable in a typical production environment

The magnitude and nature of these two activities depends upon the development environment.

1) An HP division needs to know: does the system work and can it be placed in normal production status? Can the parallel operation be discontinued? What support (including documentation) must be established before the development project can be closed?

2) An HP central group, such as ICON or CAS, needs to know: can the system be installed successfully in a typical field office? Is it a reliable and proven product? What support (including documentation and training) must be provided in the future to dependent installations? When can it be released to other offices?

3) An HP software product division, selling software to HP customers, such as MPD, needs to know: is the system performance proven? Does it meet the needs of a typical user? Can it be installed successfully, using normal support groups? What support will be required after development? What size computer configuration is required for suitable operation? When can it be released to the sales force?

The Project Leader should identify the environment in which s/he will be operating and plan the Release Phase accordingly. This chapter does not attempt to explore each environment thoroughly. but attempts to raise questions and describe situations that will prompt the Project Leader to think of his/her needs.

During the Release Phase, the development project remains open and the project team remains intact.

The subject of performance tests is worthy of note here. Sometimes, when a project enters the Release Phase, the system is subjected to a complete performance and stress test. This is done to characterize the scope of the system. This will enable prospective users to easily evaluate the system and estimate its suitability and adequacy.

However, good performance, like quality, is built into a system from the beginning of development. Checkpoint tests to measure performance and quality should be made during each phase of the development cycle, and concerns about performance should be explored when they arise. For example, performance testing during the Investigation phase can be done by using simulation techniques. Specific programs can be tested during the coding stage in order to determine their capacity. And, as mentioned above, the full system can be tested during the Release phase to determine the system's maximum capacity and configuration requirements. Additionally, product divisions require a full system performance test before the (software) product can be placed on the Corporate Price List.

Therefore, performance testing should be done continually during development and should not be limited to any one point in time.

During the Release Phase, the system may only be altered to correct errors or improve performance. No change should be allowed that alters the basic functionality of the system. If functional changes are judged necessary, the Project-Manager should be informed, and a plan of action prepared. Either the project may return to an earlier development phase (primarily to repeat the step of management review and approval), or the changes should be delayed until after the system is released and handled during the Software Support Phase (see chapter 7) procedures.

The documents that are associated with the Release Phase and described in this chapter are:

Document Name	Acronym
Support Plan (continued)	SP
Beta Test Plan	BTP
Beta Test Report	BTR
Release Phase Review and Sign-Off	

6

PROJECT ADMINISTRATION

Project administration includes procedures that initiate, direct, monitor, review, and approve work performed for an information system project. If these procedures are used effectively, they provide adequate control over project activities.

As mentioned before, a project notebook should be maintained throughout the life of the project. As each phase begins, a plan for that phase should be prepared and added to the project notebook contents.

In the earlier phases, project administration was primarily concerned with establishing the needs of the user, what the system should do to satisfy those needs, and how the system should be designed. During this phase, project administration is concerned with finding a suitable beta test site, assuring that an effective test environment is maintained, controlling the test costs, and assuring that system support personnel will be ready to receive the system.

During this phase, the original project team will be working with other departments, functions, divisions or companies (customers) depending on the environment. Additionally, the project team may be expanded to include others from these different areas. The beta test site might require sizeable expenditures for travel, lodging, and equipment which will need to be estimated, approved, and controlled.

Management should review the Release Phase plan from two points of view. First; the assignments should be realistic. This is important since the new project participants will probably be from other departments or organizations. They must be able to accept, understand, and perform their new assignments.

Second, procedures for the expanded project team should be established to report the beta test progress, report new problems, and resolve or fix problems. The success of the test depends on gathering complete, factual information and distributing that information to those responsible for fixing problems and reviewing the test results.

Support Plan Implementation

The Support Plan, prepared during the previous phase, consists of several categories of support. The plan (actual support) must be completed as per the schedule of each suppor' category.

Plan for the Beta Test

The purpose of the Beta Test Plan is to establish the requirements for the test before a site is selected. This will help with the selection of the site (described below). For example, the plan should state the volume of transactions that represents a typical user. Later, site candidates would be measured against the volume to judge their qualifications.

A sample table of contents for the Beta Test Plan is shown at the end of the chapter. The Project Leader should prepare a table of contents for the Beta Test Plan using the sample as a guideline.

This plan should be reviewed and approved by management (see section VII of the table of contents) before proceeding with the selection of the beta test site.

Select the Beta Test Site

The specific process of selecting a beta test site depends upon the development environment of the project (see introduction to this chapter). If it is necessary to select a distant user for a test site, the process will be more involved and the need to have management actively participate will be greater.

A general approach of the process is listed below and includes several steps. This list may be used as a guideline:

- o Prepare the plan and requirements for the test, and obtain management approval (see Plan for the Beta Test above).

- o The Project Leader should identify site candidates using the information in the Beta Test Plan. Each should be surveyed to obtain information related to the requirements contained in the Beta Test Plan.

- o The Project Leader should evaluate the site candidates. They should be measured against the requirements and any other criteria that are pertinent. Some of the concerns and questions that should be considered in the evaluation of the different candidates are:

- Will they provide a good test of the software product?

- Which site will benefit the most from the system? Do they need the system? Do they want it?

- Who can provide technical assistance? Who will likely be prepared to support the installation?

- Whose schedule comes closest to ours?

- Is our rapport with them conducive to a test environment?

o The Project Manager should select a site and direct negotiations. The negotiations should define the specifics of the test: technical aspects, schedule, financial agreements, equipment, and personnel responsibilities. If the site is not within HP, the legal department should participate at each step of the negotiations.

A more complete list of items that should be included in the negotiations may be found in the sample Table of Contents at the end of this chapter.

o The Project Manager should obtain management approval for the test site agreement.

Monitoring the Release Phase

The Project Leader should monitor the progress of the assignments to be sure that the work being done is going according to plan. Checkpoint meetings should be held periodically, perhaps every two weeks.

During this phase, there may be several assignments being worked on independently. These include the different categories of support and the beta test. Each of these should have their own schedule and should be tracked separately. If there are significant deviations from the plan, adjustments should be made and approved by management.

Completing the Release Phase

The development project is closed after the Release Phase is complete. Therefore, the Release Phase Review and Sign-Off has obvious significance.

o Management believes the system is suitable for use by the intended user(s)

- The system is available for "normal" distribution and use
- The system will be maintained by ongoing support activities
- The cost of development is at an end, and the cost of support begins; responsibility for system costs may change

To prepare for the Release Phase Review and Sign-Off, the Project Leader should distribute the Beta Test Report, which contains the test results, and the status of support schedules to attendees prior to the meeting. They should be asked to review the documents in advance. Optionally, other documents such as product documentation may be distributed or made available to those who request them.

The key to the final review is the results of the beta test and the preparedness of the support groups to assume responsibility for the system.

When approved, the Release Phase Review and Sign-Off form will signify the end of the development project and the beginning of ongoing support.

DOCUMENT EXPLANATION

There are several documents that are associated with the Release Phase. They are:

Document Name	Acronym
1) Support Plan (continued)	SP
2) Beta Test Plan	BTP
3) Beta Test Report	BTR
4) Release Phase Review and Sign-Off	

These documents are used in several ways. They serve as a checklist of items for the project team to perform, they serve as a plan that management must review and approve, and they record the results of project activities that management will read before the final review meeting of the development project. Additionally, they are used by new members of the project team and will orient and direct them in their assignments.

Please note that sample Tables of Contents for all documents may be found at the end of this chapter. The samples are designed for a typically large project and are, therefore, necessarily detailed. When preparing the project plan, the project leader should review the samples and tailor them to the specific project. In so doing, items pertinent to the project should be selected from the sample outlines and, if needed, unique items should be added. The format of the documents should be maintained as closely as possible to the samples in order to assure consistency between projects and different HP entities.

Each document is discussed in detail on the following pages. The major sections from each table of contents, contained at the end of this chapter, are included with the discussion. If desired, the reader may turn to the samples while reading this section to obtain a more complete understanding of the material.

Document 1 - Support Plan (continued)

The Support Plan was begun during the previous phase. It encompasses a "complete" list of support items that will be required for the software after its development is complete and it is released for normal use. The major sections of the plan are listed below:

SUPPORT PLAN
(major sections)

I) Project Identification
II) Product Documentation
III) Training
IV) Sales/Distribution
V) User Support
VI) Software Support

Each section in the plan contains a schedule for the preparation of the actual support items. The completion of these support items must be done according to each schedule.

Document 2 - Beta Test Plan

The beta test is the culmination of the development effort. If the system passes this test, it almost assuredly will be released to its intended users and used in a normal production mode (post development). This test *commits* the system to production use for the first time. By way of comparison, the alpha test, performed during the last phase, proved that the system functioned properly. It was not used in a production capacity by the user. It now must stand up to the "real world" of production.

Because of its importance, the beta test should be planned before it begins, which is the purpose of this document. The Beta Test Plan consists of seven major sections.

BETA TEST PLAN
(major sections)

I) Project Identification
II) System Test Requirements (what to test)
III) Profile Requirements of Site (define typical user)
IV) Schedule Requirements
V) Criteria for Test Termination
VI) Budget Guidelines
VII) Management Approval

The plan reflects management's opinions about the requirements for the beta test and the major criteria for selecting the test site.

This plan will be used by the Project Leader and any others who will be involved in the site selection process (please refer to the Project Administration section of this chapter). It defines the requirements for testing and the guidelines for the budget and schedule. It also indicates how the site should ideally be supported: with project team members at the beta site, with project team members located at their own location and connected to the beta site via modems, and other guidelines.

Because of the important nature of the test and its potentially large expense, management must approve the Beta Test Plan before further action is taken.

Document 3 -Beta Test Report

Before the system is released, management must have confidence that the system will perform reliably for typical users and that it can be supported by existing or planned support groups. The results of the beta test are the basis for management to form their opinion.

The Beta Test Report is written after the Beta Test Plan. It defines the specific objectives for the test before it begins. It reports the results of the test, including an evaluation of the system from the viewpoint of the test site management and the project management.

The Beta Test Report consists of 5 major sections.

BETA TEST REPORT
(major sections)

I) Project Identification
II) Site Description
III) Test Site Objectives
IV) Results
V) Evaluation

Sections I, II and III are prepared before the test begins. Their content should accomodate the requirements and guidelines contained in the Beta Test Plan. These three sections include a schedule of events and a definition of how the test will be terminated. Termination will occur because the test was successful, because the performance of the system failed to meet expectations and the test was not successful, or because the test site personnel failed to conduct the tests properly.

Section IV contains the results of the test. The completeness of this section depends on the effectiveness of the procedures used to conduct the test. This issue is discussed in the Project Administration section of this chapter.

The evaluation of the test provides for reports from different project participants. This variety will be important to management when they prepare for the phase review.

Document 4 - Release Phase Review and Sign-Off

The purpose of this one page document is to record the results of the meeting when management and other interested parties review the results of the Release Phase activities.

The approval of this final review signifies that management believes the system is acceptable for use by the intended users and that the system can be supported by existing or planned support groups.

The approval closes the development project.

DOCUMENT PREPARATION

DOCUMENTS *STEPS*

1) Support Plan

- o Complete support as per plan

2) Beta Test Plan

- o Define what should be tested
- o Define site profile
- o Establish schedule
- o Define criteria for terminating test
- o Provide budget guidelines
- o Approve plan

3) Beta Test Report

- o Describe selected test site
- o Establish test site objectives
- o Conduct test and record results
- o Evaluate test results

4) Release Phase Review and Sign-Off

- o Conduct the phase review meeting
- o Obtain release approval
- o Turn over system to support groups
- o Close project

How to Complete the Support Plan

The Support Plan was begun during the previous (Construction) phase and contains several categories of support: product documentation, training, sales/distribution, user support, and software support. Each category in the plan includes a schedule of what is to be done and when it is to be completed.

Some support items must be completed before the alpha test while others must be completed before the beta test. The remainder must be completed by the end of the project.

Therefore, the completion of the support must be done according to the schedules in the plan. Some ideas on preparing product documentation are contained in the previous chapter. A definition of how to prepare each of the other support categories is not included.

How to do the Beta Test Plan

The planning of the beta test is an essential first step in a sequence of beta test activities. To wit: if you don't know where you want to go, then how will you know when you've arrived?

The different sections of the plan (see sample Table of Contents) may be prepared by software, user, quality assurance, or support personnel according to what is appropriate for each section.

Step 1 - Define what should be tested
(BTP, section II)

The list of items in section II of the sample Table of Contents represents a checklist of items that may be tested. It includes system functions, data and volume, performance, documentation, training, and length of test or number of business cycles.

Each item in the list should be reviewed and defined for the system being tested. The documentation and training should not be overlooked for testing. Even if the system works "perfectly," it will not succeed without effective documentation and training. This is an excellent time to test the effectiveness of these items.

Step 2 - Define a profile of site
(BTP, section III)

Define the type and size of the organization. It should represent the intended, typical, and future users of the system.

Describe the minimum level of experience with similar applications that is required for an "organization" to qualify as a site candidate. Also, define the minimum level of resources (management, staff, H/W, S/W) required and the extent of their responsibility and participation. This section is an essential in evaluating test site candidates.

Step 3 - Establish schedule
(BTP, section IV)

Prepare a general calendar of events. This is another essential item that will be used to evaluate test site candidates. After the site is selected, a schedule specific to that site will be prepared and it must be consistent with the schedule defined in this section.

Step 4 - Define criteria for terminating test (BTP, section V)

The project team should define some criteria for terminating the beta test before the site is selected. The information in section V of the sample Table of Contents contains some items for consideration.

This step should include the participation of management because the information may be critical when the test is ready to be concluded. The completion of a beta test typically causes financial responsibility to shift from the developer to the user.

Step 5 - Provide budget guidelines (BTP, section VI)

Budget guidelines should be established and include travel by the project team, what the site will be expected to pay for, who will pay for delivery of new hardware or software, and the price of the system to the user after the test.

Step 6 - Approve plan (BTP, section VII)

The Project Leader should arrange for management to review and approve the Beta Test Plan before proceeding with the site selection. For a description of the site selection please refer to the Project Administration section of this chapter.

How to do the Beta Test Report

The Beta Test Plan and the selection of the beta test site must be completed before this report is prepared (see Project Administration section, this chapter). Sections II and III of this report involve planning the activities for the site. Sections IV and V record the results and evaluation of the test.

Step 1 - Describe selected test site (BTR, section II)

Describe the selected test site. This includes its name, location, contacts, and a brief profile of the organization.

The resources that will be available for the test should be identified. This includes staffing, hardware and some of the responsibilities that are assigned to the staff.

Step 2 - Define test site objectives (BTR, section III)

This section of the table of contents is a checklist of items that should be considered in preparing a specific site plan. The hardware and software configuration should be defined as well as the parts of the system that will be tested. The participants should be identified with their responsibilities.

After this is done, a schedule of events should be made. Its contents should include: training, hardware delivery, conversion (parallel runs), start up, and system (cycle) operations. The schedule should also list the individuals responsible for the events.

The procedures for reporting progress should be defined. Should they be verbal or written? At what frequency? How should problems be documented? Who should receive the information?

A schedule of reviews should also be prepared. It should include project participants, users, and management. Remember, the review process will be the means for evaluating the progress of the test and of knowing if the test is going well or not. The review schedule should not be haphazard. It should be planned in advance so that appropriate action can be taken to correct any problems that are not resolved quickly.

The criteria for terminating the test should also be defined. The Beta Test Plan contains guidelines for the test site. Note that the test acceptance criteria will be the basis for measuring the success of the test activities and evaluating the results. This criteria is of major importance and should be agreed upon before the test begins.

Step 3 - Conduct test and record results (BTR, section IV)

Section IV will be easy to prepare if section III was prepared throughly. It simply is a means of recording the results of the test, as they occur. Each item should relate to the items in section III if at all possible.

There are several items in this section: calendar of events, testing details, log of operations, and description of training and documentation used. The "reason for ending the test" will appear first among the list of items but, of course, it would be written last. The reason is because the story is not a "who-done-it" and the readers will most likely want to know why the test was terminated before reading all the details of this section.

Step 4 - Evaluate test results
(BTR, section V)

The first item of importance is to describe any significant changes to the system that were made as a result of the test.

The rating of the system according to predefined criteria should be done next. Some criteria for consideration are: reliability, performance, friendliness, and effect on other systems.

Evaluate the ability of the system to meet the test site's needs. If there are problems, they should be defined as being typical user needs or unique user needs.

Evaluations from users, project managers, and beta test site managers should be included in this section. The evaluations do not have to be consistent. The obvious parts of the system, training, and documentation should be evaluated: was the documentation adequate, acceptable, and clear? If not, should it be changed before the next installation?

How to do the Release Phase Review and Sign-Off

The Beta Test Report and other documents should be distributed to management and others who will be participating in the phase review meeting. Sufficient time for study of the documents should be allowed before the review meeting.

The purpose of the Release Phase review is to determine if the software (system) is ready to be distributed to targeted users (customers). Some questions that should be discussed at the review meeting are as follows:

- o are there any production or operations problems?
- o what were the results of the beta test?
- o does the system meet the needs of the beta test site? will they continue using the system?
- o was the system altered as a result of the beta test? are there any unresolved conflicts?
- o will the system meet the needs of the targeted user(s)?
- o how much computer resource is needed for a typical user? are there many prospects that don't have the recommended hardware configuration?
- o did the beta test uncover any new problems?
- o is the system supportable?
- o what is the status of the system support?

When the review meeting is complete, and the software system has been judged acceptable for release, the Project Leader should prepare the Sign-Off document, write the conclusions of the meeting, obtain management and user signatures, and file the original in the project notebook.

SAMPLE TABLES OF CONTENTS

The following pages contain sample Tables of Contents for all the documents described in this chapter except for the Support Plan. Please refer to chapter 5 for the Support Plan sample Table of Contents.

1) Beta Test Plan
2) Beta Test Report
3) Release Phase Review and Sign-Off

The Tables of Contents are detailed and designed for a typically large project. Because every project is unique, the Project Leader should review the Tables of Contents and tailor a new set for the specific project. The pertinent items from the samples should be selected for the new set, unnecessary ones rejected, and unique items should be added.

BETA TEST PLAN

I) PROJECT IDENTIFICATION

- A) Project name, mnemonic and project number
- B) Project abstract
- C) Project personnel (including test site personnol)

II) SYSTEM TEST REQUIREMENTS
(what to test)

- A) System functions
- B) Data and volume
- C) Performance
- D) Friendliness
- E) Documentation
- F) Training
- G) Length of test or number of business cycles
- H) Supportability (how easy to report, locate, patch and fix bugs)

III) PROFILE REQUIREMENTS OF SITE

- A) Type and size of organization
- B) Experience with application
- C) Resources (management, staff, H/W, S/W)
- D) Availability and responsibility of resources

IV) SCHEDULE REQUIREMENTS

V) CRITERIA FOR TEST TERMINATION

- A) General statement of conditions
- B) Key event(s) or time
- C) What will site do (use) after test?
- D) How will site be supported after test? (when will "normal" support activities begin? Role of development team?)

VI) BUDGET GUIDELINES

VII) MANAGEMENT APPROVAL

BETA TEST REPORT

I) PROJECT IDENTIFICATION

A) Project name, mnemonic and project number
B) Project abstract
C) Project personnel (including test site personnel)

II) SITE DESCRIPTION

A) Beta test site name, location, contacts
B) Site's business environment (size, organization)
C) Assigned resources (staff, hardware)

III) TEST SITE OBJECTIVES

A) Hardware and software configuration
B) Participants and their responsibilities
C) Master schedule of events and assignments (training, hardware deliveries, documentation, conversion, start up, operations schedule)
D) Schedule of reviews (project team, users, management)
E) Procedures to report status and problems
F) Problem correction procedures
G) Criteria for ending test and removing system
H) Criteria for user ending test and accepting system

IV) RESULTS

A) Reason for ending test (did user accept system)
B) Calendar of actual events (training, operations, reviews)
C) Testing details (compare to section II, part A above)
D) Log of system operations (data volume, response times)
E) Log of problems (time, severity, correction, consequences)
F) Description of training and documentation

V) EVALUATION

A) Significant changes resulting from test
B) Rating of "specified" system (reliability, performance, friendliness, relationship to other systems)
C) Ability of system a meet test site's needs
D) Recommendation for changes (if any)
E) Rating against abort and completion criteria (see IV above)
F) Beta site's user(s) report(s)
G) Beta site's management report(s)
H) Project manager's report

RELEASE PHASE REVIEW AND SIGN-OFF

I) PROJECT IDENTIFICATION

A) Project number, date ______________________/__/__

B) Name ______________________________

C) Project Manager ______________________________

D) Project Leader, location ______________________________

E) Intended users ______________________________

II) STATEMENT OF APPROVAL

All relevant questions and concerns have been resolved to our satisfaction. It is felt that the project is complete and that the system should be released.

III) APPROVAL SIGNATURES (if conditional, attach statement)

A) Functional Manager ______________________________

B) Project Manager ______________________________

C) User Management ______________________________

D) Users ______________________________

IV) REJECTION SIGNATURES (attach reasons)

______________________________ ______________________________

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

SISTEMAS DE INFORMACION

(DOCUMENTACION BASICA)

FEBRERO, 1984

i. SISTEMAS DE INFORMACION (DOCUMENTACION BASICA)

MODELOS

Los modelos son representaciones de la realidad.

La palabra modelo es usada como sustantivo, adjetivo y verbo, en cada uso tiene un significado especial.

Como sustantivo " Modelo" es una representación, en el sentido con el cual un arquitecto construye un modelo a escala de un edificio o un físico construye un macromodelo de un átomo.

Como adjetivo " Modelo " implica un grado de perfección o idealización, como el referirse a un estudiante modelo.

Como verbo " Modelo " significa demostrar, ejemplo: modelar un vestido.

Los modelos científicos tienen todas estas acepciones, son representaciones de estados, objetos y eventos, son idealizaciones en el sentido de que son menos complicados que la realidad y entonces son más fáciles de usar para propósitos de investigación, son más fáciles de manejar que el objeto real.

La simplicidad del modelo respecto a la realidad estriba en el hecho de que sólo las propiedades relevantes del original son reproducidas, por ejemplo en un modelo del sistema solar las esferas que representen a los planetas y al sol no necesitan ser de la misma materia que éstos.

Los modelos científicos son usados para acumular y relacionar conocimientos acerca de diferentes aspectos de la realidad, son instrumentos que sirven para explicar el pasado, el presente y para controlar o predecir el futuro.

Los modelos científicos son de tres tipos:

ICONICOS

ANALOGICOS

SIMBOLICOS

Los modelos icónicos son grandes o pequeñas representaciones a escala de estados, objetos o eventos, representan las propiedades con solo una transformación de escala. Por ejemplo: Las fotografías, dibujos, mapas.

Los modelos Icónicos del sol y sus planetas tales como los que se instalan en los planetarios son a escala reducida, en tanto -

que los del átomo (por ejemplo, el de Bohr) son a escala amplificada. En general*los Modelos Icónicos son específicos, concretos, y difíciles de manejar para fines experimentales.

Los modelos Analógicos utilizan un conjunto de propiedades para representar a otro. Por ejemplo, las líneas de contorno en un mapa son analogías de elevaciones, un sistema hidráulico puede utilizarse como un análogo de sistemas Eléctricos, Económicos o de Tráfico. Generalmente los modelos analógicos son menos específicos, menos concretos, pero más sencillos de manipular que los -- Icónicos.

Finalmente los modelos simbólicos que utilizan letras, números - y otros tipos de símbolos para representar las variables y sus relaciones, de aquí que sean el tipo de modelo más general y abs-- tracto.Normalmente son los más sencillos de manejar experimentalmente. Los modelos simbólicos toman la forma de relaciones matemáticas (casi siempre ecuaciones o desigualdades) que reflejan la estructura de lo que representan.

La ciencia emplea los tres tipos de modelos, en general se usan modelos Icónicos y Analógicos como preliminares a los simbólicos.

* En este caso, los flujos de agua representan flujos de corriente eléctrica.

SISTEMA

En un sistema -desde el punto de vista funcional- sus propiedades esenciales se pierden cuando sus partes se consideran separadamente. C.W. Churchman, propone una definición operativa para - los sistemas y, establece las condiciones necesarias para que un sistema " DELTA" sea concebido como tal:

1) "DELTA" es TELEOLOGICO, es decir, tiene una finalidad tanto en sus partes como en su conjunto.

2) "DELTA" tiene una medida de eficiencia, o de comportamiento.

3) Existe un beneficiario o cliente cuyos intereses o valores son servidos por "DELTA" de tal manera que, mientras más - "ALTA" sea la medida de eficiencia, los intereses son mejores servidos, es decir, el beneficiario es el estandar de la medida de eficiencia.

4) "DELTA" tiene componentes teleológicas que producen la medida de eficiencia de "DELTA"

5) "DELTA" tiene un ambiente definido ya sea de una manera teleológica o ateleológica que también coproducen la medida de eficiencia de "DELTA"

6) Existe por lo menos un tomador de decisiones*, quien, por sus recursos puede producir cambios en la medida de eficiencia de los componentes de "DELTA", y, por lo tanto, cambiar la medida de eficiencia de "DELTA".

7) Existe un diseñador, que conceptualiza la naturaleza de "DELTA", los conceptos del diseñador producen acciones en el T. de D. y, por lo tanto, cambios en la medida de eficiencia de "DELTA"

8) La intención del diseñador, es cambiar "DELTA" para maximizar el valor de "DELTA " al cliente.

9) "DELTA" es estable con respecto al diseñador en el sentido de que exista una garantía ímplicita de que la intención del diseñador es realizable.

NOTA:

* T. de D.

Las preguntas serían: ¿Cómo lo comprobarán ? ¿ De qué forma lo harán? ¿Qué puntos checarán ?.

Deberán estudiar las situaciones dadas, identificando:

1) TOMADOR DE DECISIONES.

Obviamente puede haber más de uno, escogieron uno de ellos para poder construir el ambiente y especificarán el tipo de relaciones que existen entre los posibles tomadores de decisiones.

2) Determinarón precisamente el ambiente (Dado el T.de D.).

3) Determinarán los clientes o beneficiarios, formando una jerarquía de beneficiarios según su importancia para el diseño propuesto.

4) Se estructuraron los componentes del sistema, comprobando que realmente sean teleológicos.

5) Para obtener la medida de eficiencia del sistema deberán considerar el tipo de insumos y productos que se tienen.

6) Finalmente, determinar si su diseño es a corto, mediano o largo plazo, especificando el tiempo en años.

SISTEMAS DE INFORMACION

Los términos datos e información se emplean a menudo indistíntamente, si bien se refieren a conceptos diferentes.

Los datos son entes del lenguaje, matemáticos o simbólicos, sobre los que generalmente se está de acuerdo para representar -- personas, objetos, acontecimientos y conceptos.

La información es el resultado de modelar, organizar, convertir y en general, elaborar datos, de tal manera que aumente el nivel de conocimientos del perceptor su comprensión de la realidad.

Aunque dato e información son conceptos distintos, están muy relacionados; la información se produce a partir de los datos. Empleando terminología de fabricación podríamos decir que el dato es la materia prima y la información del producto terminado.

Un Sistema de información puede definirse como : Un instrumento lógico que se emplea para producir información que sirva de soporte a las actividades de una organización, a partir de un conjunto de datos que son elaborados de acuerdo a unas reglas pre-establecidas.

Esquemáticamente puede representarse asi:

CONJUNTO DE DATOS   MODELO   SISTEMA DE INFORMACION

Frecuentemente un sistema de información se asocia con los diferentes niveles del proceso de información de una organización, tales como se muestra en la figura No. 1, donde se representa - la estructura conceptual de un sistema de información integrado para una organización.

El individuo que hace uso de un sistema de información es el usuario, que necesita la información como ayuda para la toma de decisiones o simplemente para realizar sus actividades. La información que proporciona un sistema de información será eficaz en tanto lo sea para el usuario.

Se presentan a continuación las principales características de los sistemas de información, que permiten evaluar en algunos casos la bondad del sistema.

SOPORTE: Define que el sistema sea manual o informatizado.

FLEXIBILIDAD: Capacidad de incluir posibles modificaciones futuras en las políticas de gestión, operación e información de la organización sin alterar su estructura básica.

MODULARIDAD: Posibilidad de ser implantado de forma gradual y de crecimiento.

NEUTRALIDAD: Sin repercución significante en la organización a causa de las características técnicas de su diseño.

ADAPTABILIDAD: Grado de adaptación a las necesidades de información de los usuarios y a los objetivos del sistema.

FIABILIDAD: Medida en base a la fiabilidad de la información producida.

EFICIENCIA: Medida en base a la eficacia de la información que proporciona al usuario.

INTEGRABILIDAD: Accesibilidad de los datos en todos los procesos evitando su reintroducción.

SISTEMAS INFORMATICOS

De acuerdo con la definición inicial de sistema de información, este es un instrumento lógico, que puede ser implantado tanto sobre medios manuales como informáticos.

El gran avance que se ha producido en los últimos años en la - concepción y diseño de los sistemas de información, se debe,-- sin embargo, a que puede ser implantado sobre medios informáticos, por lo que, si bien el estudio de un sistema de información debe ser previo al estudio del sistema informático sobre el que implantarlo, debe existir un alto grado de coordina---- ción e interrelación entre ambos.

Un sistema informático es un conjunto de hardware (material) y software (logical) básico que sirve de soporte y vehículo, - como un todo, al sistema de información. El diseño general de un sistema informático debe definir la arquitectura, el funcionamiento de un sistema informático e identificar todos sus componentes.

La Arquitectura de un sistema informático puede ser obtenida a partir de la arquitectura de información, y refleja los princi pales flujos de información entre las diferentes unidades organizativas de la empresa u organismo, tal como lo muestra la figura 2.

A partir de esta arquitectura es posible definir el funcionamiento del sistema informático. Dentro del sistema informático es - conveniente, a su vez, realizar una estrucuración en subsistemas y funciones como se hace en un sistema de información.

Los componentes en que normalmente se estructura un sistema informático son los siguientes:

- Sistemas de Información
- Bases de Datos
- Comunicaciones
- Hardware (material)
- Software (logical)

Es necesario resaltar que desde el punto de vista de un sistema informático, los sistemas de información son simplemente componentes o elementos del mismo, a los que se asocian un conjunto - de requerimientos del usuario y características generales que --

permiten el diseño del sistema informático.

El componente Base de Dátos debe definirse a partir de las clases de datos identificados durante el estudio del sistema de información y de la estructura lógica de los datos.

El componente Comunicaciones comprende todo el sistema de comunicaciones y frecuentemente los elementos de control del sistema informático.

El componente Hardware comprende todos los elementos físicos del sistema informático.

Por último, el componente Software Básico, comprende todos los elementos lógicos, a excepción del sistema de información y las bases de datos.

SISTEMA INFORMATICO GERENCIAL (MIS)

Un 'MIS' puede definirse de la manera siguiente:

Un instrumento lógico que proporciona a la dirección de una organización la información necesaria para controlar actividades, medir rendimientos, detectar tendencias, evaluar alternativas, tomar decisiones y tomar medidas correctoras.

El 'MIS' aplica a todos los niveles de dirección y ha de estar diseñado para satisfacer los objetivos de la organización.

De acuerdo con la figura 3, pueden considerarse en un 'MIS' seis - niveles; cuatro orientados al usuario, Planificación, Control, Revisión y Modelización, y dos tecnológicos, Sistema Informático y Administración de Datos.

El nivel de Planificación tiene una función extremadamente importante: La preparación de planes. Sin la existencia de un plan no es posible controlar, preveer, modelar, comprar, fabricar o vender.

En una organizacíon deben preparase varios planes (estratégicos, de recursos y operativos), debiendo establecer una jerarquía entre ellos. Los planes deben ser preparados, evaluados y entonces seleccionados los más adecuados.

El Nivel de Control está asociado con el proceso de comparar resultados con planes, poner de manifiesto desviaciones y realizar los ajustes necesarios.

El control es una extensión del proceso de planificación. El control mide los resultados de la implantación de los planes. Si los resultados no se ajustan al plan, deben aplicarse las medidas correctoras y en caso necesario revisar el plan.

El Nivel de Previsión sirve de soporte a los dos niveles anteriores en la realización de sus funciones. La previsión consiste básicamente en determinar el futuro a partir del pasado.

La previsión es el estudio de series temporales de datos, a partir de los cuales se determinan tendencias y ciclos, mediante técnicas estadísticas o de cualquier otro tipo.

Por último, el Nivel de Modelización sirve de soporte a los tres niveles anteriores en la realización de sus funciones.

Consiste básicamente en coger los modelos definidos en los otros tres niveles, asegurar la compatibilidad entre los mismos, y prepararlos de manera que puedan producir resultados cuantitativos - útiles.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

EL ENFOQUE SISTEMICO

(DOCUMENTACION BASICA)

FEBRERO, 1984

ii. EL ENFOQUE SISTEMICO
(DOCUMENTACION BASICA)

INTRODUCCION

En la medida que continuemos automatizando los procesos que controlan nuestra vida -equipo médico, tráfico aereo, generación - de energía- estaremos sujetos a confiar, cada día más en el funcionamiento de su prolífera masa de programas. Ingeniería de la Programación de computadoras (Software Engineering), es la rama de la informática que estudia la producción de esta programación, de tal manera que resulte costeable y suficientemente confiable. En consecuencia, se trata de una disciplina que debemos establecer con claridad y ejercer con eficacia y calidad.

Por programación entendemos no solamente el código, sino también la documentación requerida para desarrollar, operar y dar mantenimiento a ese código. De esta manera, debemos enfatizar la necesidad de considerar la generación de documentación oportuna, - como una actividad integrada al proceso de desarrollo de siste-- mas. Si combinamos ésto con la idea de ingeniería, tenemos que Ingeniería de la Programación es el establecimiento y uso de --- principios ingenieriles, con el objeto de obtener programación -

económica y útil para máquinas computadoras.

Por otra parte, Control de Calidad tiene como objetivo garantizar que todos los productos terminados cumplan o excedan los requerimientos de comportamiento y aceptación que han sido mutuamente -- acordados entre el usuario y quien desarrolla la programación.

Los objetivos que persigue la Ingeniería y Control de Calidad en la programación son, entre otros, los siguientes :

a) Desarrollar sistemas útiles, es decir, que satisfagan ampliamente las expectativas de todos los usuarios con problemas semejantes.

b) Desarrollar sistemas de bajo costo, lo que significa, que se ajusten a un presupuesto establecido.

c) Desarrollar sistemas seguros, o sea, sistemas protegidos - contra daños de terceras personas.

d) Desarrollar sistemas modulares, o sea, sistemas divididos - de acuerdo a funciones definidas.

e) Desarrollar sistemas adaptables, es decir, sistemas que pueden ser facilmente modificados de acuerdo a nuevos requerimientos.

f) Desarrollar sistemas portables, es decir sistemas compatibles a varios equipos de cómputo y por lo tanto facilmente intercambiables.

EVOLUCION EN EL DESARROLLO DE SISTEMAS

Originalmente el desarrollo de sistemas de programación, se iniciaba sin una exhaustiva fase de planeación y definición de requerimientos. Se trataba de producir sistemas sin considerar que éstos, en un futuro, deberían ser integrados a un sistema generalizado de información. Los problemas posteriores de integración hacian que el sistema, en el mejor de los casos, trabajará ineficientemente.

Es común encontrar que el 55% de los errores en un programa son causados durante la fase de análisis de requerimientos, de los cuales el 50% recien son detectados en la fase de construcción o desarrollo (figura 1)

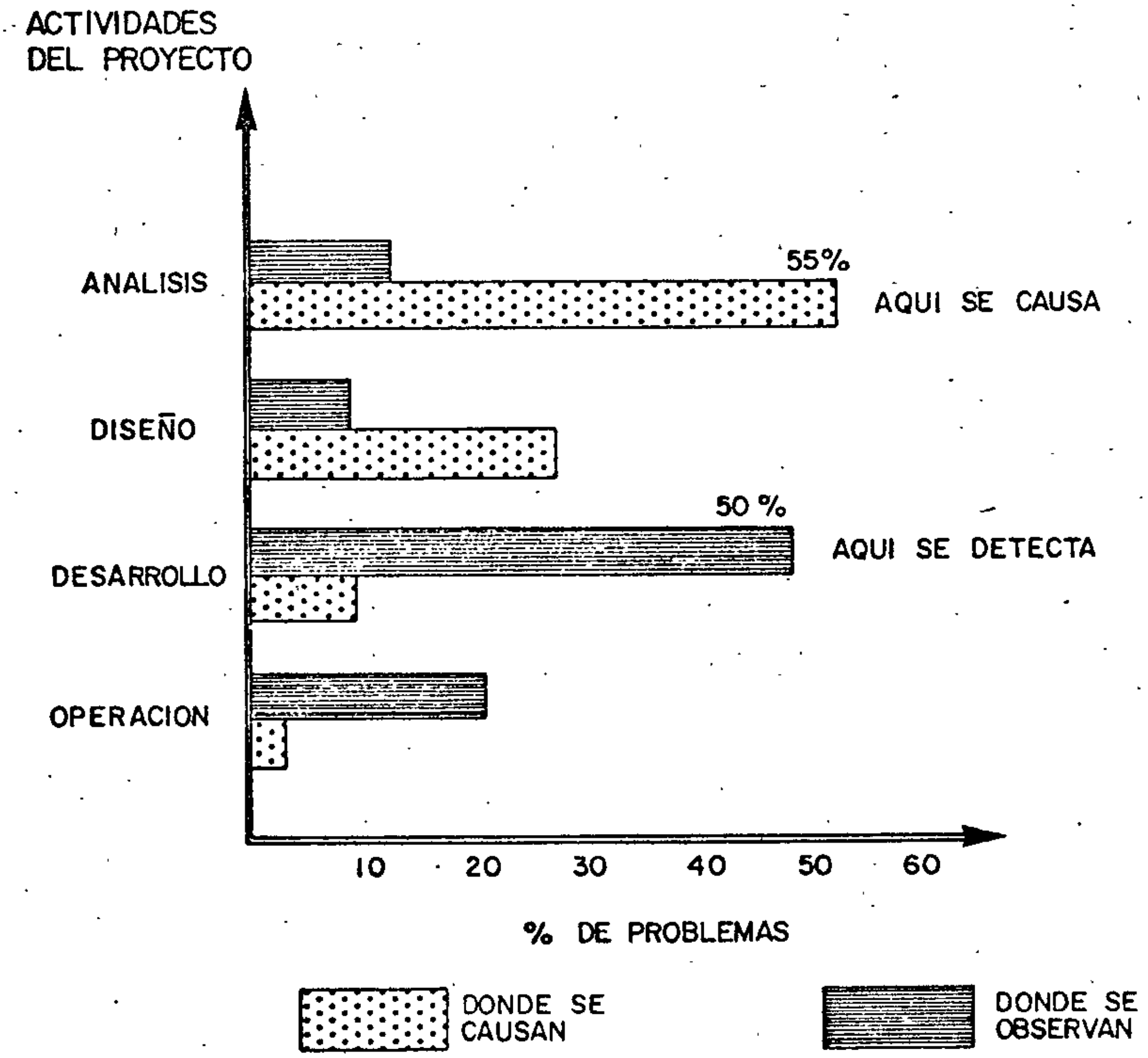

ACTIVIDADES
DEL PROYECTO
ANALISIS
DISEÑO
DESARROLLO
OPERACION
55%
AQUI SE CAUSA
50 %
AQUI SE DETECTA
10
20
30
40
50
60
% DE PROBLEMAS
DONDE SE CAUSAN
DONDE SE OBSERVAN

Figura 1

La antigua tendencia todavía no totalmente erradicada, de iniciar lo más pronto posible la escritura de código, lleva a programas cuyos errores se detectan en fases muy adelantadas en el desarrollo del proyecto, siendo el costo de su corrección muy elevado.

El costo cada vez creciente de programación hace necesario seguir un Enfoque Sistemático para su construcción y uno de prueba y error, como venía haciéndose hasta la fecha.

EL ENFOQUE SISTEMICO

El desarrollo de un sistema de programación, debe de seguir una serie de fases interrelacionadas.

Por ejemplo:

- Planeación del Proyecto
- Definición de Requerimientos del Usuario
- Arquitectura del Sistema
- Construcción de programas
- Liberación de Sistemas

Durante cada una de estas fases, uno o más productos, (por ejemplo, documentos, y/o programas de computadora) son generados. -- Las actividades desarrolladas durante cada fase y los documentos y programas producidos deberán de cumplir con las normas, estándares, revisiones y auditorias que se realicen. La relación entre las fases, los productos y las revisiones se muestran en la Figura 2.

A continuación se resumen los objetivos de cada una de estas fases y se señalan los productos finales que deben de obtenerse en cada una de ellas.

El objetivo básico de la fase de planeación es desarrollar un plan para la administración y el desarrollo técnico del proyecto. Durante esta fase se definirán los requerimientos técnicos del sistema en forma resumida, así como la descripción de la documentación, los productos a producir, las actividades y las revisiones a realizar, durante el transcurso del proyecto. Estándares y normas deben ser preparados, discutidos y publicados. Deberá establecerse un mecanismo para controlar el progreso del proyecto e - implantar acciones correctivas, en caso necesario. Por último, - se deberá establecer los requerimientos de una "Biblioteca de Soporte de Programas ", la cual provee control y visibilidad sobre -

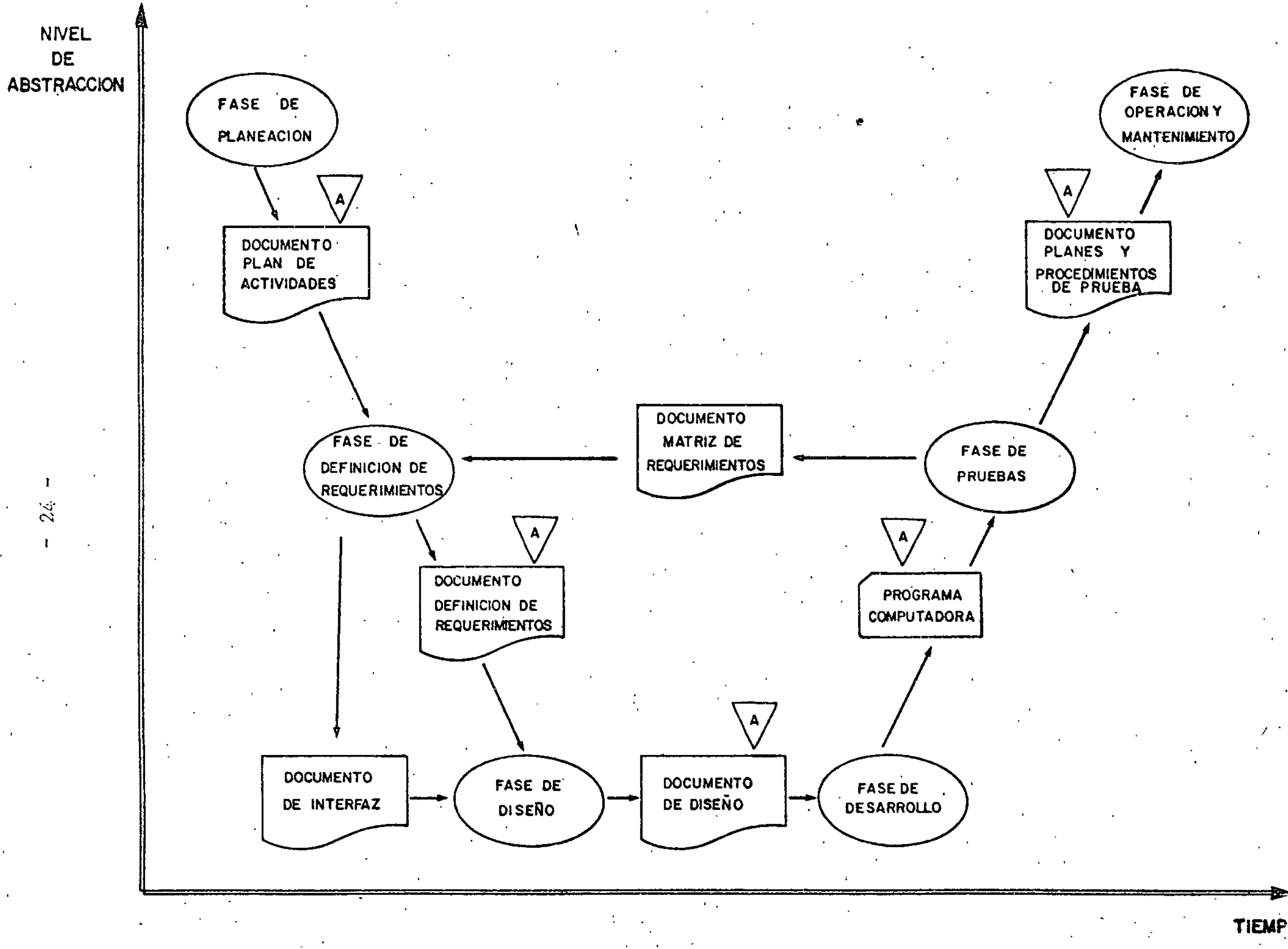

NIVEL DE ABSTRACCION
FASE DE PLANEACION
A
DOCUMENTO PLAN DE ACTIVIDADES
FASE DE DEFINICION DE REQUERIMIENTOS
DOCUMENTO MATRIZ DE REQUERIMIENTOS
FASE DE PRUEBAS
A
DOCUMENTO PLANES Y PROCEDIMIENTOS DE PRUEBA
FASE DE OPERACION Y MANTENIMIENTO
A
DOCUMENTO DEFINICION DE REQUERIMIENTOS
A
PROGRAMA COMPUTADORA
DOCUMENTO DE INTERFAZ
FASE DE DISEÑO
A
DOCUMENTO DE DISEÑO
FASE DE DESARROLLO
TIEMP

Figura 2

el avance del proyecto.

Los productos finales de esta fase pueden ser, entre otros, los siguientes:

- Matriz de requerimientos
- Ruta crítica de actividades
- Establecimiento de recursos
- Normas y estándares de documentación

La fase de planeación es constante durante todo el proyecto y asegura a la dirección que los planes y aspectos técnicos del proyecto hayan sido satisfechos.

Una vez observada la factibilidad del proyecto, se procede a iniciar la fase de Definición de Requerimientos la cual tiene como objetivos, primero, el delinear y describir las características funcionales y parámetros de comportamiento del sistema a nivel detallado, lo cual permite posteriormente el diseño del mismo; y segundo, designar el criterio y los métodos para verificar que el sistema desarrollado cumple con los requerimientos establecidos.

El producto final de esta fase es un documento de Definición de Requerimientos de Programa de Computadora y deberá de ser revisado y aprobado por el usuario final. Aunque esta fase termina con la -- edición de este documento, subsecuentes cambios y nuevos requeri--- mientos van a aparecer; esto va a traer como resultado una actualización de los requerimientos, función que debe ser estrechamente - vigilada y oportunamente documentada.

Continuando con el proceso de la programación, el objetivo de la - fase de Arquitectura es el traducir los requerimientos del programa de computadora, definidos en el documento anterior, en un diseño detallado, bien definido y lógicamente estructurado. El docu-- mento de requerimientos de programa de computadora documenta las - funciones que el programa de computadora debe de realizar, utili-- zando para ello una combinación de prosa en lenguaje natural, --- gráficas y notación matemática.

Para fines de este documento, podemos definir al diseño de programas de computadora como el proceso de transformar una serie de instrucciones en lenguaje natural a una serie de instrucciones en -- pseudo-código. Según el lineamiento fundamental del diseño de programas de arriba hacia abajo, los requerimientos tienen que divi--

dirse en partes que resulten fáciles de entender; por lo tanto, es importante definir las principales funciones que se encuentran implítas en los requerimientos a fin de diseñar un programa de computadora para cada una de ellas. La"Arquitectura" de programas de - computadora precisamente identifica esas funciones y las asocia a un programa.

Una vez definidos el número de programas que van a desarrollar sus interfaces asociadas, se procede a elaborar un "Diagrama de Estructura" para cada programa, mismo que representa la estructura jerárquica y funcional de cada uno de ellos. Los módulos resultantes - del diagrama de estructura de cada programa representan una sub- - función. La especificación del proceso de cada módulo es descrita utilizando las técnicas de " Programación Estructurada" en un lenguaje de alto nivel denominado "Pseudo - Código" estructurado, que permite independizar el diseño del lenguaje final de cómputo.

La fase de Arquitectura termina con la edición de un documento de Diseño de Programa de Computadora y deberá, como en la fase ante-- rior, ser revisado y aprobado por el usuario final.

Una vez traducidos los requerimientos del usuario a pseudo-código la siguiente fase es la de Construcción. Los objetivos principales de la fase de construcción son la codificación de los módulos, la verificación del correcto desempeño de cada módulo, la integración de los módulos para formar programas, la integración de programas para formar sistemas; así como, asegurar que el sistema de programación funcione correctamente de acuerdo a los requerimientos del usuario.

La fase de construcción se considera terminada cuando el usuario acepta los resultados de los programas de computadora integrados en un sistema. La condución de las pruebas, previa revisión y -- aceptación de los planes y procedimientos por parte del usuario - final, marcan el final de esta fase. Desde luego, esta fase es - iterativa y continúa hasta que todas las modificaciones y discrepancias generadas por las pruebas hayan sido corregidas.

Los principales productos finales de esta fase son código fuente lanzadores, módulos objeto, listados de programa, resultados de - las pruebas de construcción, integración y aceptación, y en caso necesario, las versiones actualizadas de los documentos descritos anteriormente.

Finalmente, la fase de Liberación del Sistema es la última de las fases del proceso de programación, y es en donde se reflejan los aciertos y errores de las fases previas. La fase de Liberación de la programación se inicia con la primera instalación del sistema integrado, una vez que la programación haya sido aceptada - por el usuario final en base a los documentos de planes y procedimientos de prueba.

Se identifican aciertos en la medida en que los requerimientos - de los programas de computadora satisfacen las necesidades del usuario. Asímismo, gran parte del éxito de la programación radica en la metodología que se emplee para controlar los cambios que se tengan que realizar a los programas después de su primera instalación.

La documentación final de esta fase normalmente es llamada Manual de Usuario y permite a los operarios utilizar la programación en forma cabal y eficientemente, según las especificaciones del u-- suario vertidas en el documento inicial de requerimientos.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

CONTROL DE CALIDAD

(DOCUMENTACION BASICA)

FEBRERO, 1984

iii. CONTROL DE CALIDAD
(DOCUMENTACION BASICA)

INTRODUCCION

Debido a la frecuente insatisfacción por parte de los usuarios con respecto a los sistemas que se desarrollan, es necesario contemplar un Programa de Control de Calidad con el objeto de garantizar que todos los productos terminados cumplan los requerimientos de compartimiento y aceptación que han sido mutuamente acordados entre el usuario y quien desarrolla el sistema de información.

Las cualidades que debe tener un sistema se enumeran a continuación:

a) Seguro.- El producto realizará las funciones requeridas por el usuario bajo condiciones normales.

b) Probado.- El producto final deberá ser probado para garantizar la correcta satisfacción de los requerimientos del usuario.

c) Mantenible.- Un producto es mantenible cuando es fácil de entender por los programadores de mantenimiento, es fácil de modificar y probar cuando existan nuevos requerimientos, se rectifiquen deficiencias y se corrijan errores.

d) Eficiente.- Un producto es eficiente cuando satisface los requerimientos del usuario sin exceder los recursos destinados.

3) Entendible.- Un producto es entendible cuando pueden dominarse fácilmente las funciones del producto y las relaciones con otros productos. Es conveniente recalcar que el producto debe ser entendible también por las personas que no están rela-

cionadas con el producto.

f) Portable.- Un producto es portable si las características del diseño incluyen independencia de dispositivos, de tal manera que el producto funcione con las mismas características en otro sistema de cómputo.

Estas cualidades, adicionando que el sistema debe producirse dentro del tiempo y costo planeado, son difíciles de satisfacer dada la problemática en el desarrollo de sistemas entre las que se consideran las características heterogéneas de la gente que está involucrada en el proyecto, la deficiencia técnica y comunicativa del usuario, aunado a la del equipo de cómputo, originando de esta manera una baja calidad en los productos desarrollados.

Para lograr un control de calidad adecuado deben definirse una serie de normas o estándares válidos en el desarrollo de los sistemas y monitorear los productos generados por medio de revisiones y auditorías para que dichos productos cúmplan con las normas establecidas. Otro factor importante para el logro de un adecuado control de calidad es la utilización de herramientas automáticas en el desarrollo de los sistemas, ya que permiten la normalización y automatización en la producción de los productos.

El grupo de control de calidad deberá ser organizado como una función separada del grupo que desarrolla, teniendo la suficiente autonomía y autoridad para implementar un adecuado programa de control de calidad, el que deberá incluir:

a) Las especificaciones de los objetivos y alcances del programa.

b) La especificación detallada de procedimientos para el control de calidad.

c) La especificación detallada de las normas para el desarrollo de los sistemas.

d) Auditorías internas y revisiones periódicas por expertos calificados.

e) Estudio y elaboración de herramientas automáticas para el cumplimiento de las normas.

Las características que debe tener el Programa de Control de Calidad son las siguientes:

a) Factibilidad.- El programa deberá ser tanto técnica como administrativamente factible.

b) Actual.- El programa deberá estar basado en las metodologías utilizadas en la actualidad.

c) Actualizable.- El programa deberá ser fácilmente actualizable para incluir nuevas metodologías de software.

Para concluir, puede considerarse que la parte medular del Programa de Control de Calidad está compuesto del establecimiento de las normas, las auditorías, revisiones y el estudio y elaboración de herramientas automáticas para el cumplimiento y apoyo de dichas normas.

NORMAS O ESTANDARES

Dado que en el ciclo de desarrollo de sistemas tienen que generarse una serie de productos que deben poseer una cierta calidad, dichos productos deben elaborarse de acuerdo a ciertas normas.

Para la elaboración de las normas es conveniente tomar en cuenta que debe tenerse una amplia experiencia en lo que va a normarse.

Las normas que sean realizadas deberán considerarse apropiadas por el grupo de personas que las utilizará, ya que de otra manera probablemente serán ignoradas.

La palabra estándar puede significar:

a) un requerimiento obligatorio

b) una convención o práctica recomendada

c) un lineamiento

En el caso de desarrollo de sistemas puede aplicarse cualquiera de las 3 acepciones generándose el siguiente espectro en la estandarización:

Tipo de Reglas	Ninguna	Guías de estilo	Guías de estándares	Estándares rígidos
Método de Reforzamiento	Ninguno	Pláticas Estructuradas	Revisiones y Htas. Autom	Htas. Autom.
Grado de Reforzamiento	Ninguno	Flexible	Firme	Rígido
Analogía	Anarquía	Democracia	Soberanía	Dictadura

Además las normas deben ser actualizadas ya que generalmente son estáticas. En algunas ocasiones no sólo son obsoletas sino que además crean conflictos con otras.

Por otro lado, en la elaboración de las normas debe tenerse la habilidad para ser reforzadas a través de la automatización, lo que permitirá, entre otras cosas, aumentar la productividad, facilitar la tarea de revisión, obtener una mayor aceptación por parte de los integrantes del grupo, y todo ello con la consecuente reducción del costo total de la función de control de calidad.

A continuación se proponen algunas políticas para el control de calidad en las diferentes fases del ciclo de vida de los sistemas. Las normas o estándares serán detallados en los capítulos siguientes.

1) Definición de Requerimientos:

a) Los requerimientos deberán ser escritos de manera jerárquica a fin de ser fácilmente localizados, accesados, mantenidos y probados.

b) Deberá definirse cuál es el contenido del documento de especificación de requerimientos, así como su organización con el objeto de elaborar diferentes secciones en paralelo.

c) Definir quién o quiénes revisarán y aprobarán el documento de especificación.

d) El documento deberá incluir los recursos de cómputo necesarios para operación y para pruebas de mantenimiento.

e) Deberá establecerse el procedimiento para la actualización del documento.

f) El documento diferenciará los requerimientos a corto, mediano y largo plazo.

2) Arquitectura:

a) Definir cómo va a ser escrito y almacenado el diseño.

b) Deberá definirse cuál es el contenido y organización del documento.

c) Deberán explicarse a manera de ejemplos las posibilidades de extender, contraer o adaptar el documento y los resultados esperados.

3) Construcción:

a) Deberá definirse el lenguaje de programación a utilizar, teniendo en mente la facilidad que éste ofrece para transportar programas de una computadora o sistema operativo a otro, sin perder las características intrínsecas de cada sistema y a su vez, facilitando la traducción del detallado de los módulos realizados anteriormente.

b) En la selección de algunos lenguajes habrá la necesidad de establecer las estructuras estándar de programación -sin las extensiones del lenguaje- que serán permitidas con el fin de aumentar la transportabilidad del mismo.

c) La documentación de un programa la constituye la claridad de su estructura, por ello deberá definirse la documenta--

ción, los comentarios dentro del programa y el ėstilo de programación que estimulen una programación mantenible y auto-documentada, evitando la necesidad de instrucciones de flujo arbitrariamente complicadas, y estableciendo guías y recomendaciones para ello.

d) Por último, deberá definirse el medio ambiente, así como la organización y contenido de los documentos, de planes y procedimientos de pruebas.

4) Liberación:

a) Deberá definirse la documentación mínima necesaria que se generará durante el ciclo de vida, y que servirá como guía para la operación, la utilización y el mantenimiento de los sistemas.

REVISIONES Y AUDITORIAS

En el ciclo de vida de los sistemas se cometen una gran variedad de errores. Conforme va avanzando la realización del proyecto, la detección y corrección de estos errores se hace más costosa. Es por esto que se utilizan auditorías y revisiones para detectar estos errores; para una mayor efectividad de estos métodos, es conveniente calendarizar una serie de ellas durante el ciclo de vida y sobre cada producto generado.

Las revisiones y auditorías pueden ser realizadas por el humano o por la computadora; en el mejor de los casos deben combinarse ambas.

Existen dos tipos básicos de revisiones realizadas por el humano:

a) Inspecciones o revisiones estáticas, y
b) recorridas (walkthroughs) o revisiones dinámicas.

La revisión estática puede ser formal o informal, y es importante hacer notar que el propósito de la revisión es encontrar errores, mas no corregirlos.

Por otro lado, las auditorías deben verificar que los controles internos funciones efectivamente y que se cumplan las normas y estándares establecidos.

La auditoría dentro del ámbito de sistemas se enfoca básicamente en la determinación de las deficiencias y las mejoras en métodos, formas de control y de la utilización de recursos en el sistema.

HERRAMIENTAS AUTOMATICAS

Dentro del ciclo de vida de los sistemas es muy importante investigar y elaborar herramientas automáticas, ya que facilitan la labor de control de calidad, además de garantizar una estandarización de los productos generados. Con este fin se estudian las herramientas automáticas para conocer qué tipos existen, cuáles son sus características, qué ventajas y/o desventajas representa su uso y evaluar el apoyo que pudieran proporcionar en el control de calidad y en el ciclo de vida de los sistemas.

Ventajas de las Herramientas Automáticas.-

Las ventajas principales de las herramientas automáticas (H.A.) son:

a) Aumentar la productividad.- Se incrementa la productividad ya que la generación de los productos es más rápida.

b) Facilitar las tareas.- Las tareas que se realizan son más sencillas ya que se cuenta con el apoyo y experiencia de manera indirecta por parte de las personas que realizaron la herramienta, y con el apoyo directo de dicha herramienta.

c) Incrementar la creatividad.- Se incrementa la creatividad ya que la persona que desarrolla tendrá más tiempo para investigar, sin perderlo ahora en labores repetitivas.

d) Mejorar la calidad.- Dado que las herramientas generan productos con las normas establecidas, se propicia por ende una determinada calidad en dichos productos.

e) Estilo único.- No se tendrá una variedad ilimitada de estilos, lo que ahorraría tiempo en el entendimiento de los productos.

f) Mayor aceptación de las personas involucradas.- Las herramientas automáticas son imparciales, por lo qué no tienen preferencias sobre personas en particular.

Desarrollo Automático de los Ambientes.-

La noción de un desarrollo automático de los ambientes y de una colección de herramientas computarizadas para apoyar a los sistemas de información está incluida en la Administración de Proyectos en Informática.

El objetivo principal es proporcionar a quienes desarrollan software, herramientas y técnicas que les permitan mejorar el proceso de producción de software y la calidad del mismo; los ejemplos más comunes de estas herramientas son: editores de textos, compiladores, rutinas para copiar archivos, rutinas de impresión.

Algunos Problemas en el Uso de las Herramientas de Software.-

Hay con la mayoría de estas herramientas algunos problemas:

1) Fallan en el apoyo de la metodología del desarrollo de software o en la captura de cualquier dato que ayude en el control del proceso del desarrollo de software. Existe la necesidad de mejorar la calidad de las herramientas para satisfacer este problema.

2) No apoyan al ciclo de vida del desarrollo de software en su totalidad, apoyan principalmente actividades de codificación; hay, pues, necesidad de más herramientas para especificación de software, diseño y pruebas, así como para la administración de proyectos, además es preciso integrarlos en una metodología pues están encaminados a una etapa específica del proceso de desarrollo; a la especificación del problema (análisis), al diseño global o detallado, a la implementación o a la evaluación, etc.

Además de lo anteriormente señalado existe:

- Falta de compatibilidad.- Las herramientas son difíciles de combinar entre sí.
- Falta de uniformidad.- El conjunto de herramientas disponibles difieren de acuerdo a las máquinas, sistemas operativos y lenguajes, haciendo difícil el pasar de un ambiente de desarrollo a otro.
- alta de adaptabilidad.- La mayoría de las herramientas están diseñadas para usarse en forma predeterminada y no es fácil de utilizarlas para distintos patrones.

Propiedades y Capacidades requeridas de las Herramientas.-

En la consideración de apoyo automático a metodologías para el desarrollo de sistemas de información, hay que tomar en cuenta la estructura en la cual la herramienta automática funcionará, especificando sus propiedades generales, entonces podrá procederse a identificar las herramientas que pueden ser de mayor utilidad en cada caso.

La metodología para desarrollo de software involucra la necesidad de nuevas herramientas, las cuales es conveniente que posean las características que a continuación se enuncian:

. Propósito singular.- Cada herramienta deberá realizar una función perfectamente definida, o bien un conjunto de funciones estrechamente relaciondads entre sí pero definidas.

. Fácil de usarse.- No deberá necesitarse tener gran cantidad de documentación para poder hacer uso de la herramienta, ésta deberá ser compacta y accesible.

. Consistencia con otras herramientas.- Las herramientas deberán interactuar entre sí de manera consistente manteniendo interfases estándar y protocolos de comunicación.

. Adaptabilidad.- Deberán adaptarse a los requerimientos específicos por el usuario, por ejemplo: pueden operar en varios modos para apoyar a distintas clases de usuarios. Las herramientas deben proveer de un conjunto de "omisiones" que pudieran ser modificadas para satisfacer las necesidades individuales del usuario, en este sentido cada usuario puede empezar con una herramienta genérica y puede ajustar parámetros y opciones a su propio modo de uso.

. Inteligencia local.- Puede proveerse a una herramienta de esta capacidad deseñándola para reunir información útil y que sea almacenada en una base de datos privada, quizá usando un sistema manejador de Base de Datos como herramienta para este proceso.

. Apoyo para el ciclo de vida del software.- Las herramientas deben ser desarrolladas para apoyar el ciclo de vida del software desde el análisis de requerimientos hasta la prueba y evolución a través del tiempo. Las herramientas deben proporcionar no sólo soporte técnico para una fase específica del ciclo de vida, sino que deben facilitar la asistencia de la dirección así como el pasar de una fase a otra dentro del ciclo de vida.

. Apoyo para la dirección de desarrollo de software.- Dos clases de dirección son esenciales:

- dirección o manejo de la configuración del software, y
- dirección del personal en el desarrollo de software.

El manejo de la configuración puede lograrse por medio de un conjunto de herramientas que permitan la captura de la estructura del programa, satisfacer los requerimientos de interfase, solucionar problemas de reportes, así como actualizar versiones fuente y objeto de los programas.

Asimismo, puede obtenerse fácilmente información referente a la productividad individual y a la cantidad de esfuerzo asociado a cada una de las partes del proyecto de software y estar organizada ésta por proyecto o por módulo.

Tal información es particularmente valiosa para conocer mejor el proceso de desarrollo de software, lo que puede llevar a la utilización de mejores técnicas para estimar y evaluar el esfuerzo requerido para nuevos proyectos y a la identificación y consecuente desarrollo de mejores herramientas de software.

Herramientas para desarrollar Sistemas de Información.-

Algunas herramientas que pudieran ser parte del medio ambiente para desarrollo de sistemas de información son las siguientes:

. Sistemas operativos.
Provee acceso a recursos compartidos, interfase estándar a dispositivos de I/O, facilidades de carga, encadenamiento de programas.

. Procesadores de lenguaje.
Compiladores y/o intérpretes, junto con el apoyo del "run time" son necesarios para todos los lenguajes de programación que están siendo usados en el medio ambiente de desarrollo de sistemas de información.

. Editores de textos.
Una herramienta de este tipo es de gran ayuda para la documentación asociada con el desarrollo del sistema, así como para la codificación de los programas. Algunos ambientes de programación pueden contar con editores de sintaxis directa que son capaces de checar el programa de entrada, de acuerdo a las reglas de sintaxis del lenguaje.

. Formateadores.
Los programas formateadores son útiles para la documentación y para la programación, ayudan en el proceso de tablas, ecuaciones y textos, y ofrecen medios para producir documentos.

Aceptan textos de programas como entrada y producen textos de programas reformateados como salida.

. Analizadores dinámicos.

Es capaz de proporcionar información acerca de un programa durante su ejecución, puede obtenerse una "instantánea" dando los valores de las variables del programa, un "rastreo" que muestra las unidades del programa que han sido llamadas, un análisis del tiempo de ejecución dedicado en cada unidad de programa o una contabilidad del número de veces que algunas líneas específicas han sido ejecutadas; con un graficador puede observarse la ejecución dinámica del programa.

Toda esta información es útil para recuperación de errores y aumentar la eficiencia del sistema. Analizadores semejantes existen para sistemas manejadores de base de datos, con las que es posible contar el número de operaciones de E/S, o de páginas recuperadas de memoria auxiliar a memoria principal.

. Administración de la configuración.

Una herramienta para administrar la configuración puede ser usada para asegurar la continuidad de la documentación asociada con el proyecto de desarrollo de un sistema de información, incluyendo una o más versiones del código fuente y objeto.

Esta herramienta es un elemento importante para controlar cambios durante el desarrollo del sistema de información.

. Herramientas de registro de bitácora.

Estas pueden ser usadas por sistemas de auditoría como un modo de determinar el uso de varios programas en el ambiente del sistema de información. Por medio de éstas pueden registrarse los accesos del usuario al sistema y brindar ayuda en la recuperación de cualquier violación a la seguridad del sistema.

Está lista no se considera completa, sólo se ejemplifican algunas herramientas que pueden apoyar el desarrollo efectivo y con gran calidad de sistemas de información. Existen algunas más, encaminadas a facilitar el análisis, el diseño de la arquitectura, el diseño detallado de base de datos, verificación de programas, administración de proyectos, así como mantenimiento de bibliotecas, entre otros.

Mejores herramientas para desarrollar sistemas de información se tendrán si apoyados en experiencias del pasado se integran éstas en una metodología, pues ésta permitiría:

a) Unificar acciones de la dirección y procedimientos técnicos con que se cuenta y cubrir así completamente el ciclo de vida de los sistemas.

b) Contrôlar los costos de desarrollo y evolución aumentando la productividad del personal con la ayuda de herramientas automáticas.

c) Establecer guías para la medición de la calidad del software desde su inicio y a través de todo su ciclo de vida.

Otras Herramientas.-

Para el adecuado control de calidad existen un par de herramientas que combinadas pueden ser de gran ayuda:

a) Analizador Estático de Código.

Dentro de los estándares se contemplan los de codificación que servirán para que los programas sean fáciles de leer, entender y modificar. Para llevar a cabo la auditoría de estos estándares, puede realizarse manual o automáticamente.

Realizarlo manualmente sería por medio óptico de una persona, la cual se encargaría de encontrar las violaciones a los estándares.

Realizarlo automáticamente se haría por medio de un programa de computadora -que contendría los estándares-, en caso de encontrar violaciones mandaría un mensaje de aviso. Este tipo de herramienta se le conoce como analizador estático de código.

Otras de sus funciones es examinar el texto de un programa para determinar si hay algunas variables de programas que son usadas antes de que se les asigne un valor, o si hay variables a las que se les ha asignado un valor y nunca fueron usadas, o si hay segmentos de "código muerto", que no pueden ser descubiertos por medio del flujo de control de programa.

Una vez que el programa sea examinado y probado puede considerarse que es de una calidad excelente.

b) Bibliotecas de código reusable.

Es un programa que maneja de manera automatizada el código, la documentación y los casos de prueba generados durante el desarrollo para ser accesados nuevamente, sin necesidad de ser nuevamente desarrollados.

Una vez que los módulos desarrollados tengan una calidad revisada por medio del analizador estátivo de código, y probados, son una fuente valiosa y reusable. Por lo general los módulos o programas tienen que ser nuevamente programados con la consecuente pérdida de tiempo, ya que no existen los medios adecuados de disponibilidad e información de lo generado previamente.

El código que tenga calidad y en muchos casos es reusable, deberá ser colectado en una biblioteca de código reusable. La promulgación de políticas y procedimientos sobre el código reusable es un medio de alcanzar las metas del programa de control de calidad con el mínimo costo y tiempo. El trabajo de tener una biblioteca es un trabajo que no es sencillo de realizar, por lo que deberá tenerse una persona que atienda a dicha biblioteca, lo cual será de gran beneficio.

Para la creación de la biblioteca existen dos métodos básicos para identificar las fuentes de código reusable dentro de una organización: revisar las bibliotecas existentes que son usadas por muchos programadores y procesar un cuestionario que identifique el código útil que será entregado a la biblioteca.

Una vez que el software reusable es seleccionado para una biblioteca, la documentación de apoyo debe ser preparada. Los puntos siguientes se consideran importantes:

Operación. Un ejemplo del método de invocación incluyendo un conjunto conveniente de argumentos formales.

Objetivo. Una corta explicación del fin de código utilizando los argumentos formales del ejemplo de uso.

Limitaciones. Las condiciones que deberá tener el código para que se asegure su funcionamiento.

Advertencias. Cualquier demanda irrazonable sobre los recursos; la documentación de un error conocido pero todavía no reparado.

Módulos requeridos. Identificar los módulos que deberán estar presentes al momento de invocar el código.

Argumentos. Por cada argumento formal dado en el ejemplo de uso, definir su tipo y propósito.

Autor. El nombre y organización que desarrolló el código original, así como la fecha.

Algoritmo. Si el algoritmo no es usual o complejo, una breve mención de él o una mención de su procedencia.

Registro de modificaciones. Por cada modificación hecha al código, dar el nombre del autor que hizo el cambio, la fecha y el propósito de la modificación.

Para la utilización del código -que previamente debe estar aprobada su calidad- es recomendable hacerlo por medio de la biblioteca de código reusable.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

PLANEACION DEL PROYECTO
(DOCUMENTACION BASICA)

FEBRERO, 1984

iv. PLANEACION DEL PROYECTO
(DOCUMENTACION BASICA)

CURSO ADMINISTRACION DE PROYECTOS EN INFORMATICA 1984

INTRCDUCCION

La sabiduría es la habilidad de ver con mucha anticipación las consecuencias de las acciones actuales, la voluntad de sacrificar las ganancias a plazo corto, a cambio de mayores beneficios a largo plazo y la habilidad de controlar lo que es controlable y de no inquietarse por lo que no lo es. Por — tanto, la esencia de la sabiduría es la preocupación por el futuro. No es el mismo tipo de interés en el futuro que tienen los videntes, que sólo tratan - de predecirlo. El sabio trata de controlarlo.

La planeación es proyectar un futuro deseado y los medios efectivos para conseguirlo. Es un instrumento que usa el hombre sabio; mas cuando lo manejan personas que no lo son a menudo se convierte en un ritual incongruente - que proporciona, por un rato, paz a la conciencia, pero no el futuro que se - busca.

La necesidad de planear el desarrollo de proyectos es tan obvia y tan - grande, que es difícil encontrar alguien que no esté de acuerdo con ella. Pero es aún más difícil procurar que tan planeación sea útil, porque es una de las actividades intelectuales más arduas y complejas que confronta el hombre. No hacerla bien no es ningún pecado, pero sí lo es contentarse con hacerla - menos que bien.

La necesidad de llevar un control estricto en el desarrollo de proyectos, es tan importante que requiere de una metodología que se apege a cualquier -

proyecto de desarrollo de un sistema de información. Sin duda alguna la primera fase en el proceso de desarrollo de sitemas de información es la de planeación.

El objetivo básico de la fase de planeación es elaborar un plan para la administración y el desarrollo técnico del proyecto.

Esta fase de planeación no debe de consumir más de dos por ciento del costo total de desarrollo. Esto es con el objeto de prevenir al usuario de invertir una considerable cantidad de recursos y dinero, sin conocer la factibilidad del proyecto.

LAS PARTES DE LA PLANEACION

La planeación debería ser un proceso continuo, y por tanto, ningún plan es definitivo; está siempre sujeto a revisión. Por consiguiente, un plan no es nunca el producto final del proceso de planear, sino un informe "provisional". Es un registro de un conjunto complejo de decisiones que actúan una sobre otras y que se puede dividir de muchas maneras distintas. Cada planificador tiene distinto modo de subdividir las decisiones importantes, las diversas maneras de dividir un plan en partes son generalmente cuestión de estilo o de preferencia personal. Por eso no necesitamos ocuparnos de las ventajas o desventajas relativas de las diferentes formas de dividir un plan.

El orden en que a continuación se dan las partes de la planeación, no representa el orden en que se deben llevar a cabo. Volvamos de nuevo la atención sobre el hecho de que el conjunto de decisiones que implica la planeación no puede dividirse en subconjuntos independientes. Por tanto, las partes de un plan y las fases de un proceso de planeación al cual pertenecen, deben actuar entre sí. Es principalmente cuestión de la filosofía que sustente la planeación, la que indica qué partes están contenidas en un plan y la atención relativa de que sean objeto.

Las partes sólo se identifican brevemente en este momento porque posteriormente se analizarán las más importantes.

1. Fines: especificar metas y objetivos.

2. Medios: elegir políticas, programas, procedimientos y prácticas con las que habrán de alcanzarse los objetivos.

3. Recursos: determinar tipos y cantidades de los recursos que se necesitan; definir cómo se habrán de adquirir o generar, y cómo habrán de asignarse a las actividades.

4. Realización: diseñar los procedimientos para tomar decisiones, así como la forma de organizarlos para que el plan pueda realizarse.

5. Control: diseñar un procedimiento para prever o detectar los errores

o las fallas del plan, así como para prevenirlos o corregirlos sobre una base de continuidad.

	Planeación	Admon.	Metodología	Document.
Planeación	D	x	x	x
Definición de Req.	C	x	x	x
Diseño de Arq.	C	x	x	x
Construcción	C	x	x	x
Liberación	C	x	x	x

D - definición C - control

LA PLANEACION TACTICA Y LA ESTRATEGICA

Frecuentemente se hace la distinción entre la planeación táctica y la estratégica, pero rara vez se aclara. Las decisiones que para una persona pueden ser estratégicas, para otra, probablemente sean tácticas. Esto sugiere que la distinción es más relativa que absoluta. Efectivamente, gran parte de la confusión y aparente ambigüedad obedece al hecho de que la diferencia entre la planeación táctica y la estratégica es tridimencional.

1. Cuanto más largo e irreversible sea el efecto de un plan, más estratégico será. Por ende, la planeación estratégica trata sobre las decisiones de efectos duraderos y difícilmente reversibles; por ejemplo, la planeación de la producción de la semana siguiente será más táctica

y menos estratégica que la planeación de una nueva planta o de un sistema de distribución. La planeación estratégica es una planeación a largo plazo. La planeación táctica abarca períodos más breves. Pero "largo" y "breve" son términos relativos y, por ende, también lo son "estratégico" y "táctico". En general, la planeación estratégica se interesa sobre el período más largo que merezca considerarse; la planeación táctica, sobre el período más breve. Se necesita ambos tipos de planeación, pues se complementan. Son como las dos caras de una moneda: podemos verlas separadamente, inclusive discutirlas aparte, pero no las podemos separar en la realidad.

2. Cuantas más funciones de las actividades de una organización sean afectadas por un plan, más estratégico será. O sea, la planeación estratégica tiene una perspectiva amplia. La de la planeación táctica es más estrecha. "Amplia" y "estrecha" son también conceptos relativos que así aumentan la relatividad de lo "estratégico" y lo "táctico". Un plan estratégico para un departamento puede ser táctico desde el punto de vista divisional. Si las demás circunstancias permanecen inalterables, la planeación al nivel de la organización es generalmente más estratégica que la planeación a cualquier otro nivel organizativo inferior.

3. La planeación táctica trata de la selección de los medios por los cuales han de perseguirse objetivos específicos. Estos objetivos, en general, los fija normalmente un nivel directivo en la empresa. La planeación estratégica se refiere tanto a la formulación de los objetivos como a la selección de los medios para alcanzarlos. Así pues, la

planeación estratégica se orienta tanto a los fines, como a los medios. Sin embargo, "medios" y "fines" son también conceptos relativos; por ejemplo, "dar publicidad a un producto" es un medio cuyo fin es "venderlo". Sin embargo, "venderlo" es un medio para alcanzar otro fin: "obtener ganancias", y las ganancias a su vez son un medio para muchos otros fines.

En resumen, la planeación estratégica es una planeación corporativa a largo plazo que se orienta hacia los fines (pero no de manera exclusiva). Debería ser obvio que se necesitan tanto la planeación estratégica como la táctica para obtener el máximo beneficio.

ESTUDIO DE FACTIBILIDAD

En todo proceso de Administración de Proyectos se comienzan con una etapa de planeación para formalizar los cambios conceptuales y determinar la factibilidad de perseguir un desarrollo posterior. Las solicitudes de cambio de información y de nuevos sistemas se pueden originar de varias fuentes dentro de una organización. Por lo general, dichas solicitudes las recibe el Departamento de Sistemas, donde se determina si pueden satisfacer mediante el mantenimiento o mediante mejoras menores a los sistemas ya existentes. Si esto no es posible pasan a formar la base para iniciar el primer paso del proceso de desarrollo.

OBJETIVO DEL ESTUDIO DE FACTIBILIDAD

El objetivo del estudio de factibilidad es informar a la administración las características probables, costos y beneficios al llevar a la práctica un sistema específico y proporcionar los requerimientos generales referentes al proyecto en forma de reporte formal. Si resulta práctico, los resultados del Estudio deben ofrecer soluciones alternativas y conceptuales con los costos y ganancias relacionadas, tales como un enfoque complejo que ofrezca mejoras a largo plazo y un enfoque menos complejo con mejoras a corto plazo.

La determinación de los costos del proyecto incluye la planeación de las actividades del proyecto en las etapas subsecuentes y el cálculo de tiempo, personal y requerimientos de equipo. En este paso es crítica la participación del usuario para juzgar el efecto de las soluciones alternativas y para justificar gran parte de la evaluación económica. Además, a los experimentados analistas de sistemas, se les solicita que reúnan y analicen la información adecuada y que lleguen a conclusiones prácticas. La responsabilidad fundamental de este equipo combinado de trabajo es determinar - la factibilidad técnica y de operación del sistema solicitado y averiguar si producirá un reembolso aceptable de la inversión.

El resultado final de un Estudio de Factibilidad es la decisión del comité directivo, de aprobar la fase siguiente del proyecto. Esta decisión es muy importante porque se requiere un compromiso mayor de recursos para las siguientes etapas. Así, se debe tratar de eliminar todo proyecto que no sea factible. Si se obtiene la aprobación, el comité -- directivo asigna las prioridades en relación con otros proyectos aprobados y con los recursos disponibles.

INTEGRACION DE LOS GRUPOS DE TRABAJO.

El Estudio de Factibilidad trata también sobre los aspectos técnicos del proyecto propuesto y requiere de soluciones alternativas. Se necesitan expertos técnicos y considerable experiencia sobre sistemas para reunir la información adecuada, analizarla y llegar a conclusiones prácticas. Este trabajo técnico requiere demasiado para ser realizado por alguien que no sea un experimentado analista de sistemas.

Las decisiones operativas y técnicas que sean inadecuadas durante este paso, pueden pasarse por alto en todo el resto del proceso. En el peor de los casos, un error - de esa naturaleza podría ocasionar la terminación de un -- proyecto válido o la continuación de un proyecto que económica o técnicamente no sea factible.

El éxito de todo el proyecto depende de la calidad y capacidades de los analistas de sistemas y de los usuarios responsables del Estudio de Factibilidad.

La realización del estudio requiere de la formación o contitución de dos grupos, uno de los cuales deberá fungir como coordinador del estudio y el segundo como ejecutor del mismo, a continuación son descritas las características y funciones de ambos:

GRUPO COORDINADOR O COMITE DE DECISIONES.

El Comité de Decisiones es el mecanismo mediante el cual se hace participar a los Directivos de la Empresa en las decisiones que habrán de tomarse a lo largo del estudio. Es conveniente que dicho Comité esté integrado por personas de alto nivel, entre los cuales estén los responsables de las unidades de: Programación, Organización y Métodos e Informática.

Asimismo, el Comité deberá ser presidido por una persona de alto nivel jerárquico, nombrado preferentemente por el Director de la Empresa, de manera que pueda garantizar el acceso necesario a las distintas áreas involucradas ya que de ello depende en gran medida el éxito del estudio.

FUNCIONES:

- Definir los objetivos específicos y cobertura del estudio.
- Integrar al grupo técnico que se encargará del desarrollo.
- Proveer al grupo técnico de los elementos de apoyo necesarios para su correcta operación.

- Servir como medio de enlace entre las áreas involucradas en el estudio y el grupo técnico.
- Someter a la consideración de los niveles superiores de la Empresa los objetivos del estudio.
- Dirigir y controlar permanentemente el desarrollo del estudio, determinando las prioridades de actuación sobre el mismo y decidir de acuerdo a los resultados parciales sobre la continuación o reorientación de dicho estudio.
- Analizar y evaluar los resultados finales y propuestas del grupo técnico, decidiendo sobre las acciones y requerimientos que mejor satisfagan las necesidades de la Organización.

GRUPO TECNICO

El grupo técnico deberá estar constituido por los diferentes especialistas en disciplinas, tales como: el análisis de sistemas, procesamiento de datos, investigación de requerimientos de información, métodos e instrumentos de captación, técnicas en diseminación y flujos de información, etc., conjuntamente con los elementos interiorizados en las políticas y características actuales de operación de la Empresa.

Es recomendable que el responsable de la unidad de informática o en su defecto el de organización y métodos, dirijan al grupo técnico y que ambos participen en el Comité de Decisiones.

FUNCIONES:

- Definir las estrategias de acción para el desarrollo del estudio.
- Elaborar el plan de trabajo y programa de actividades pa

ra el desarrollo del estudio.

- Someter a la consideración del Comité de Decisiones los puntos anteriores para su aprobación.
- Desarrollar y documentar las diferentes etapas del estudio.
- Realizar la integración y síntesis del estudio, incluyendo las alternativas de solución y sus características.
- Llevar a cabo la selección y presentación de alternativas al Comité de Decisiones para su análisis y decisión.
- Implementar las soluciones adoptadas por el Comité de - Decisiones.

ELABORACION DEL PLAN DE TRABAJO DEL ESTUDIO DE FACTIBILIDAD.

La verificación y revisión constantes de las decisiones anteriores resultan esenciales para lograr una planeación de sistemas eficaz. Las tareas del proyecto y los estimados de calendarios y presupuestos deben actualizarse - continuamente en cuanto se tenga conocimiento de más pormenores. Por consiguiente, el administrador del proyecto debe reconocer que los planes de trabajo requieren de una actualización y verificación contínuas.

Durante la elaboración del plan de trabajo del Estu-- dio de Factibilidad, el equipo del proyecto delínea las tareas específicas que se requieren para ejecutar el plan de trabajo. Por ejemplo, el plan de trabajo podría especificar el número de operaciones actuales que se revisarán así como designar a los entrevistados, los entrevistadores, los temas, la duración planeada de cada entrevista, el trabajo calculado para elaborar la documentación, y las fechas específicas de inicio y terminación. El uso del plan de trabajo como transición de un paso al siguiente es un recordatorio de que el producto final de cada paso sirve como base para comenzar el siguiente.

PRESENTACION DEL ESTUDIO DE FACTIBILIDAD

Al finalizar la actividad de este estudio, se mantienen dos niveles de revisión, que sirven para el equipo del proyecto ex--- ponga sus resultados y recomendaciones tanto a la administración del usuario como a los miembros del Comité Directivo.

Las circunstancias de cada proyecto determinarán la estrategia precisa implícita en la estructura y presentaciones del Reporte Ejecutivo. Sin importar la estructura de que se trate, el Reporte Ejecutivo deberá incluir por lo menos una presentación clara y concisa de:

- El campo de acción y objetivos del Estudio.
- Los problemas y limitaciones
- El resumen de las consideraciones económicas.

Al preparar este Documento Final, el Equipo del Proyecto debe esforzarse en ser creativo al desarrollar las sugerencias específicas y al presentar pruebas que apoyen lo anterior.

En general, el Equipo del Proyecto debe evitar la presentación de una sola alternativa al Comité de Desiciones. Aunque -- por una parte las recomendaciones deben ser claras y concisas, - por otra, es importante tener la certeza de que todas las alternativas principales se presenten parcialmente de tal forma que - sea fácil hacer una comparación.

Al concluir el Estudio de Factibilidad, el Comité de Desiciones está en posición de aceptar o rechazar las recomendaciones -- del Equipo del Proyecto. Si se decide continuar con el proyecto, tomando en cualquiera de las alternativas identificadas en el Estudio de Factibilidad, el Comité de Desiciones deberá establecer las prioridades compatibles con los costos y beneficios estimados así como la fecha para iniciar la Fase de Definición de Requerimientos.

CONSIDERACIONES CLAVE

Necesidad de compromisos y decisiones formales.

Un proyecto no debe continuar sin la aprobación formal de trabajo efectuado durante el Estudio de Factibilidad y sin un compromiso igualmente formal sobre los recursos por parte de los usuarios, la organización del sistema y la adminstración general. También podría ser necesario contar con recursos externos.

En algunos casos se puede caer en la tentación de hacer cambios parciales a nivel informal; no obstante, la adopción parcial de las soluciones propuestas no implican que se acepten y aprueben formalmente las recomendaciones y resultados obtenidos por el equipo. De hecho, este tipo de enfoque fragmentario podría acabar con todo el proyecto, dado que si sólo se aceptan algunas partes de lo propuesto, es factible que su implantación no tenga éxito al fallar la coordinación necesaria con todas las partes restantes que conforman el todo. Además, para que muchas partes a nivel individual den resultados prácticos y justifiquen los costos, es necesario finalizar antes las partes consideradas como pre-requisitos.

Prioridad del Proyecto

Por lo general, cuando se inicia un estudio de Factibilidad, se espera mucho de él y sus partidarios lo defienden con entusiasmo, pero conforme transcurre el tiempo, el entusiasmo va disminuyendo y se dan casos en que algunos de los partidarios pierden interés o dedican su atención a otros intereses. Cada nuevo proyecto de sistema entra en competencia con los demás y una organización puede tener varios proyectos en proceso simultáneamente. Todos ellos rivalizan por tener los recursos de la empresa : personal, tiempo, dinero e incluso la atención de fabricantes ajenos a la empresa.

Por lo tanto, es de importancia vital que el proyecto se postergue o se le dé la prioridad que le corresponda al finalizar la Fase de Planeación de Sistemas, relacionándolo a otras que estén en proceso. La prioridad está sujeta a la reevaluación efectuada al surgir nuevos proyectos.

Sin embargo, ya que el mismo personal puede participar en más de un proyecto, hay que distribuir el tiempo para los proyectos de acuerdo a una escala de prioridades.

La administración exitosa de sistemas debe definir y dejar bien establecidad dichas prioridades antes y durante el proyecto.

EVALUACION ECONOMICA DEL PROYECTO

La finalidad de la evaluación económica es comparar el desarrollo estimado y los costos operativos con los beneficios identificados en un nivel general de precisión que permita a la administración determinar la factibilidad del proyecto.

En general, los costos de desarrollo se pueden calcular a partir de los requerimientos de personal y equipo indicados en el plan de proyecto preparados - en la actividad anterior.

Deberá incluirse asimismo el costo del personal administrativo de planta. La presentación de los costos dependerá del estilo administrativo de la organización en la que se esté implantando el sistema.

Por lo general, los costos operativos y beneficios son más difíciles de -- calcular e implican un problema más crítico. Resulta muy conveniente considerar éstos últimos de la siguiente forma:

Los beneficios tangibles son aquellos para los que el valor en pesos - se puede cuantificar y medir en forma razonable; los beneficios intangibles, por su parte, se pueden cuantificar en unidades que no sean pesos o bien se describen o identifican subjetivamente. La decisión para proceder con el desarrollo de un sistema se puede basar en todos o cualquier punto de los siguientes:

-Beneficios y costos operativos actuales.

El costo de operar el sistema en las circunstancias actuales y los beneficios esperados (cuantificados).

-Beneficios y costos futuros.

Una proyección de los costos y beneficios actuales según los cambios operativos planeados o esperados (tales como: políticas, procedimientos, volúmen y organización).

-Beneficios y costos intangibles.

Otros costos y beneficios particularmente difíciles de cuantificar,- o en los que la cuantificación aporta un poco al proceso de toma de decisiones. Estos costos y beneficios se podrían dividir más aproximadamente en medibles y no medibles.

Siempre que sea posible, se deben identificar y documentar las consideraciones intangibles. Al identificar los beneficios se debe tener mucho cuidado y certeza de que sólo los beneficios apropiados se asignan al proyecto de sistemas propuesto; muchos beneficios pueden obtenerse en una circunstancia - ya existente sin tener que desarrollar un nuevo sistema.

Por ejemplo:

Si en una planta de manufacturación tiene lugar un proceso de ensamblado semanas antes de tener el ensamblado final debido a que el inventario actual no es confiable, el proceso mencionado se puede eliminar mejorando el sistema ya existente.

Esto originaría un mejor inventario sin tener que desarrollar un nuevo sistema de planeación de requerimientos de materiales, por lo tanto, - el valor del beneficio no estaría en relación directa al desarrollo de un nuevo sistema.

La identificación y cuantificación de costos y beneficios es una parte estándar del Estudio -de Factibilidad que no deberá ser alternada por el equipo - del proyecto.

En resumen final de los costos y beneficios operativos y de desarrollo, en general es recomendable describir cada una de las alternativas a un nivel muy - detallado (incluyendo la alternativa de "No hacer nada") y en términos de comparaciones.

Por ejemplo:

La descripción económica de cada proyecto específico debe incluir una presentación completa y total de:

- Los costos de desarrollo.
- Los costos operativos cuantificables.
- Otros costos.
- Los beneficios tangibles.
- Los beneficios intangibles (medidos y no medidos)
- Hipótesis y limitaciones operativas.

La administración debe aprobar y dar prioridad a los proyectos potenciales basándose en factores como:

- El reembolso comparativo de la inversión
- Probabilidad de éxito
- Realismo de hipótesis y limitaciones

LA ESTIMACION DE COSTOS EN EL DESARROLLO DE SISTEMAS

OBJETO/RESUMEN

Este tema se refiere a la estimación de los costos en el desarrollo de sistemas de información, sus problemas y sus técnicas.

Para ello, se trata de justificar, en primer lugar, la necesidad de estas estimaciones, en relación con una adecuada gestión de los recursos y proyectos informáticos.

Posteriormente se analizan los diversos factores y problemas que afectan a esta estimaciones, buscando los puntos de similitud y diferencia con las estimaciones de otras actividades.

Finalmente se realizan conclusiones sobre las bases de las diferentes técnicas de estimación utilizadas en cada parte del ciclo de vida de las aplicaciones, se presentan algunas de las principales técnicas utilizadas.

INTRODUCCION

La estimación de costos en informática es un tema polémico. Las técnicas de estimación tienen partidarios y detractores, implican problemas que conviene conocer pero también suponen ventajas importantes.

Los objetivos de las estimaciones de costos son varios.

- Ayudar a la toma de decisiones sobre la oportunidad de aplicaciones y sobre las diversas alternativas que se plantean, tales como desarrollo propio a la medida, contratación exterior, adquisición de un paquete y variantes en la concepción de la aplicación (tiempo real, batch, base de datos, etc.)

- Ayudar a la gestión de proyectos informáticos. Las técnicas de estimación son una parte esencial de cualquier metodología de administración de proyectos.

- Ayudar a la evaluación del personal, pues, voluntariamente o no, el salario y la promoción de cada profesional informático están relacionados, en alguna medida, con la idea que la dirección tiene de sus resultados, los cuales se suelen referir a la calidad de sus aplicaciones o programas, al cumplimiento de los plazos previstos, etc., aspectos influídos por el procedimiento de estimación que, cuando no es racional, puede conducir a resultados injustos.

Así que para decidir la conveniencia de implantar aplicaciones y el procedimiento para su desarrollo o adquisición, para dirigir eficazmente un proyecto y para evitar tratamientos injustos al personal, es necesario disponer de procedimientos adecuados de estimación.

Pero, por otra parte, los aún escasos procedimientos existentes, poco fiables y de necesaria adaptación a cada instalación, pues se han obtenido de entornos concretos, son casi ignorados por los directivos de las entidades y los departamentos informáticos, quienes continúan haciendo o pidiendo estimaciones intuitivas, existan o no procedimientos, ya que la gestión no puede prescindir de la estimación.

A continuación se realizan algunas consideraciones sobre las particularidades de la estimación de costos y sus técnicas, presentándose también varias de las que han venido desarrollándose en los últimos años, a fin de ofrecer una panorámica sobre la situación actual de esta importante cuestión.

CONSIDERACIONES SOBRE LAS ESTIMACIONES DE COSTOS INFORMATICOS

El problema general de las estimaciones de costos informáticos puede plantearse de la forma siguiente:

- Se trata de predecir el costo que tendrán el desarrollo de nuevos sistemas relativos a una APLICACION o a alguna de sus partes.

- Esta estimación deberá ser realizada en alguno de los momentos de su CICLO DE VIDA.

- Los trabajos habrán de efectuarse en un PLAZO determinado, ajustándose a una cierta CALIDAD, de acuerdo con unos METODOS y mediante unos RECURSOS con unas caracterfsticas y capacidades determinadas.

- Bajo todos los condicionantes anteriores, la estimación propiamente dicha consiste en predecir las CANTIDADES DE RECURSOS que serán utilizadas, las cuales habrán de multiplicarse por sus COSTOS ESTANDAR para obtener el costo total.

- Para realizar una estimación puede aplicarse una serie de TECNICAS, pero también han de tenerse en cuenta los ASPECTOS HUMANOS que implica.

Planteada de esta forma, la estimación de costos informáticos no es muy diferente de la de los costos de cualquier -

otra obra humana, como por ejemplo la de construir un puente. Sin embargo, el hecho es que en informática estas estimaciones se realizan raramente en la actualidad y,cuando se hacen, las posibilidades de un fuerte sobrepasamiento son grandes.

LAS APLICACIONES

Las principales características de las aplicaciones informáticas que inciden en la estimación de sus costos son las siguientes:

- Falta de especificación suficientemente detallada y coherente de requerimientos en el momento en que la estimación debe realizarse. Este es un problema muy común en los desarrollos informáticos y su causa es profunda pues, desde cierto punto de vista, las distintas etapas de la construcción de una aplicación son redefiniciones, cada vez a mayor detalle de la aplicación a realizar y así una definición completa y detallada de una aplicación puede ser tan costoso y precisar de tanto tiempo como su desarrollo.

  En la construcción del puente, la situación es semejante a la del constructor que se viera forzado a estimar su costo teniendo unas ideas vagas y no definitivas sobre el espacio y profundidad a salvar, el uso al que será destinado, la anchura que debe tener, la carga que habrá de soportar, las características del suelo en que debe apoyarse, etc.

- Tamaño y complejidad son dos conceptos importantes en una estimación, pero no son equivalentes, aunque, en general, cuanto mayor es una aplicación también es más compleja. El tamaño está relacionado principalmente con la cantidad de funciones a realizar (informes escritos, entradas de datos diferentes, formatos de pantalla, tablas, algoritmos de cálculo, etc.) mientras que la complejidad se refiere a las interacciones entre las funciones.

Una aplicación sencilla puede, en principio, dividirse en partes casi independientes, mientras que en una aplicación compleja todas sus partes están fuertemente interrelacionadas, por lo que deben diseñarse de una sola vez y las modificaciones o errores descubiertos en un punto pueden tener una incidencia en el resto. Ejemplos de sistemas sencillos, aunque posiblemente grandes, pueden ser muchas de las aplicaciones batch que se realizan, mientras que son mucho más complejos los compiladores y aún más los programas de control de los sistemas operativos.

- Modificaciones sobre las especificaciones iniciales durante la construcción. Experimentalmente puede comprobarse que estas modificaciones se dan siempre, aunque sus causas son variadas: descubrimiento de "errores" en el planteamiento inicial, nuevas necesidades que van apareciendo durante el proceso de construcción, ampliaciones propuestas por el mismo personal de desarrollo en base a su mayor conocimiento del problema a resolver y del diseño del sistema, etc.

LAS ESTIMACIONES DE COSTOS

Las estimaciones de costos deben realizarse en varios momentos a lo largo de la vida de una aplicación. Algunos puntos especialmente interesantes en una institución con una correcta gestión son los siguientes:

- Al final de los estudios previos de oportunidad, con respecto a todo el ciclo (desarrollo, explotación, mantenimiento), a fin de determinar su costo/beneficio.

- Al final del análisis funcional, también con respecto a todo el ciclo, para gestionar el proyecto y volver a confirmar su costo/beneficio.

- Al especificar cada programa en relación con su desarrollo, para planificarlo y controlarlo.

- Periodicamente durante las fases de análisis orgánico y programación, respecto al trabajo y costos restantes, a fin de gestionar el proyecto.

- Al terminar la aplicación, con respecto a la explotación y al mantenimiento, para asegurar los recursos necesarios.

- Durante el mantenimiento, al especificar cada modificación, estimando los recursos necesarios para realizarla y su incidencia en la explotación, con el objeto de planificar y comprobar su oportunidad económica. Además, una vez comenzada su ejecución, deberá estimarse periodicamente el trabajo restante y su costo.

En consecuencia, se presentan los siguientes casos diferentes de estimación de costos informáticos:

1) Antes de un desarrollo (proyecto total, modificación, o - programa) determinar los recursos necesarios para realizarlo, su incidencia en la explotación, la repercusión que - tendrá en los proyectos futuros y los costos asociados.

2) Durante un desarrollo, determinar el trabajo y costos restantes hasta finalizarlo.

Las estimaciones del segundo tipo pueden descomponerse,- a su vez, en dos estimaciones, una del primer tipo (desarro-- llo total) cuya precisión va mejorándose conforme disminuyen- los supuestos, y otra que se refiere al avance conseguido hasta el momento.

EL PLAZO Y LA CALIDAD

El plazo tiene una incidencia de cierta importancia so-- bre el costo total de un proyecto, tanto si se trata de una - construcción como del desarrollo de un sistema informático. - En este último caso pueden realizarse las siguientes consideraciones:

- Hay partes del proceso de desarrollo que son necesariamente secuenciales. Por ejemplo, la especificación funcional del sistema debe estar acabada y aprobada antes de comenzar el diseño detallado, la especificación de un programa debe realizarse antes de comenzar su diseño y codificación, etc.

- Hay recursos que pueden convertirse en "cuellos de botella" en el proceso de desarrollo. Por ejemplo, el tiempo de cómputo disponible puede ser insuficiente a partir de un cierto número de programadores, resultado, si se sobrepasa, una disminución en la productividad.

- En los equipos humanos son muy importantes el proceso de aprendizaje y la intercomunicación. El primero implica tiempos improductivos que crecen proporcionalmente al número de componentes del grupo, pero la segunda es aún más grave, pues tiene tendencia a crecer más que proporcionalmente, según la fórmula:

$$\frac{n\ x(n-1)}{2}$$

- De estos dos últimos hechos se deduce la paradójica "ley de Brooks": Añadir personal a un proyecto informático retrasado lo retrasa aún más. En efecto, a partir de ciertos momentos la necesidad de aprendizaje del nuevo personal y su interacción con el resto del grupo pueden conducir a un aumento mayor del tiempo improductivo que del productivo.

En cuanto a la calidad, es evidente que también ha de influir en el costo. Pero en primer lugar habría que definir - lo que se entiende por este concepto. Yendo de lo más global a lo más detallado, las características que definirían la calidad de los sistemas informáticos podrían ser las siguientes:

1) Eficacia: Grado en que se adapta a las necesidades que dieron origen a su creación.

2) Eficiencia: Utilización mínima de recursos en su - explotación.

3) Adecuación a estandares: Grado de documentación, - legibilidad de los programas, etc.

4) Ausencia de errores en los programas.

Siendo el último de estos criterios el más concreto y generalizable, es el que ha motivado un mayor número de estudios hasta ahora. Como resultado de ellos y de la experiencia diaria, puede afirmarse que actualmente es imposible asegurar la ausencia de errores en un programa medianamente complejo. Por ello, el momento en que acaba el desarrollo de un- sistema de información y comienza su mantenimiento es relativamente arbitrario y, además, el costo del sistema crece con- el nivel de calidad que se precise.

Problemas muy semejantes en la calidad y el plazo se dan en los proyectos de construcción como el del ejemplo del puente. La diferencia consiste en que su influencia sobre el --

costo está ampliamente reconocida en las obras civiles pero no ocurre lo mismo en el desarrollo de aplicaciones, considerándose frecuentemente que:

- Mediante una buena prueba puede asegurarse la ausencia de errores en una aplicación, sin aumentar significativamente su costo.

- Es posible acortar el plazo de un proyecto informático lo necesario a base de añadir más personal, manteniendo el número total de meses-hombre que se estime emplear (es decir, el costo).

Como hemos visto, en el desarrollo de aplicaciones, igualmente que en el caso de la obra civil, ambos puntos no pueden considerarse válidos, en general.

LOS METODOS Y LOS RECURSOS

Los métodos y las herramientas que se utilizan para el desarrollo, la explotación y el mantenimiento de las aplicaciones son aún, al contrario de lo que podría parecer, muy escasos.

- En muchas instalaciones, los procedimientos de trabajo se reducen a la preparación de una documentación cuyo contenido puede interpretarse, además, con bastante libertad, quedando el resto (el "cómo hacerlo") a la iniciativa y experiencia de cada uno.

- Las herramientas son, en general, poco más que el sistema operativo, los compiladores de los lenguajes, bibliotecas y varios programas de utilería.

- Estas herramientas, son bastante complejas, precisando a veces de especialistas (técnicos de sistemas) para establecer los procedimientos de su utilización. Además, están sometidas a frecuentes cambios que dificultan aún más su conocimiento detallado por parte de quienes las van a utilizar.

-- Los recursos son principalmente de dos clases: persona y material.

- El personal disponible para realizar los trabajos relativoas a una aplicación suele tener una formación y experiencia muy variadas, aunque frecuentemente escasas en sistemas semejantes a los que se trata de desarrollar o utilizar. En informática es común que cada sistema tenga características que lo diferencian significativamente del resto y, por otra parte, lo reciente de estas técnicas, su rápida evolución y la duración de los proyectos contribuyen a que sea difícil encontrar personal realmente experimentado en casos semejantes.

- Otro problema típico del personal informático, en algunas instalaciones, es su dedicación dispersa entre un gran número de actividades, alternando, por ejemplo, su trabajo sobre un proyecto en desarrollo con la atención a pequeñas modificaciones y arreglos sobre aplicaciones en explotación. Esta forma de trabajo, si es frecuente, puede conducir a una baja importante en la productividad, con el consiguiente incremento en los costos, a causa de los tiempos de preparación de cada actividad elemental.

- Mención especial merece el caso del "Jefe de Proyecto", - quien tiene bajo su responsabilidad la dirección de su desarrollo y que con frecuencia tiene una carencia total deformácián en los aspectos gerenciales (planificación, control, gestión económica, gestión de personal, etc.) que se va a ver obligado a ejercer.

- En cuanto al material informático necesario para el desa-- rrollo y explotación de los sistemas, las dificultades surgen de la frecuente escasez de tiempo del computador para-la primera actividad y de una planificación óptima para la segunda. La disponibilidad de máquina para el desarrollo-es un factor que tiene una fuerte incidencia en el costo de los sistemas. Debe anotarse aquí la tendencia a la utilización cada vez mayor de este recurso conforme se va po- - niendo a disposición del personal de desarrollo nuevos útiles y herramientas (bibliotecadores, generadores de programas, documentadores, sistemas interactivos de preparación-de programas, etc.), tendencia que es coherente con la evolución técnico-económica que se viene produciendo y que - conduce a un creciente encarecimiento relativo del perso-- nal técnico en comparación con el material de proceso de - datos.

- Otro hecho característico de la informática es la necesa-- ria adaptación de los sistemas a su entorno concreto de explotación. Así, si es necesario desarrollar una aplica- - ción sobre un entorno diferente del final, las consecuencias pueden ser bastante negativas para la productividad y el costo total, debido a las conversiones que habrán de tener lugar.

En relación con estas características de los métodos, herramientas y recursos informáticos, y siguiendo con la comparación de la obra civil, una situación semejante se daría si el constructor del puente hubiera de realizarlo en unas condiciones inusitadas (por ejemplo sobre un glaciar) con un conjunto de artesanos (bastantes de ellos aprendices) cada uno con sus especiales habilidades e ideas, provistos de herramientas poco potentes y escasas. Puede comprenderse que esta situación es muy diferente de la idea (obra conocida, dirigida por un equipo de profesionales experimentados, dotada de abundante maquinaria y siguiendo un plan riguroso) y que, por lo tanto, en la informática no solamente sea baja la productividad general, sino que también se de una gran variabilidad entre distintas personas, proyectos e instalaciones.

CANTIDADES Y COSTOS ESTANDAR DE LOS RECURSOS

Una vez planteadas las premisas del problema: aplicación, plazo, calidad, métodos y recursos disponibles, deben predecirse las cantidades de recursos que serán necesarias para que, multiplicadas por sus costos estandares, pueda obtenerse el costo total del proyecto.

Como se ha dicho anteriormente, los recursos más significativos pertenecen a dos clases diferentes: personal y material. En principio, bastaría, en consecuencia, con deducir -

las horas-hombre y horas-computador necesarias y multiplicarlas por los costos de estas unidades. Sin embargo, el planteamiento de un sistema de estimación de costos informático para una entidad concreta precisa de un esquema algo más complejo.

En cuanto a los costos estandares, se refieren a cada unidad en que se miden los consumos de las secciones homogéneas y se establecen para cada ejercicio contable, mediante los presupuestos de gastos y consumos, con la siguiente fórmula:

$$CSi = \frac{\text{Gastos presupuestados para la sección i} + \text{Recargos de gastos indirectos presupuestados para la sección i}}{\text{Cantidad de unidades de consumo que se presupuesta cargar sobre aplicaciones para la sección i}}$$

Todos estos aspectos son semejantes a los que se dan en otras áreas, tales como la ingeniería civil. La diferencia consiste en que en estas últimas actividades suele existir una infraestructura administrativa (contabilidad analítica, presupuestos, etc.) que facilita el conocimiento de la influencia de cada factor y su cuantificación, mientras que no suele ocurrir lo mismo en los departamentos de informática.

Naturalmente, el mayor problema consiste en pasar de las premisas de aplicación: plazo, calidad, recursos y métodos, a la predicción de las cantidades que se van a consumir de cada recurso. Sobre este aspecto existen una serie de referencias y teorías, a las que puede denominarse con "técnicas de estimación" que se analizan más adelante.

La estimación de los costos informáticos no plantea solamente problemas técnicos, sino que presente algunos aspectos-humanos que merece la pena tener en cuenta al enfrentarse a - este tema. Estos aspectos son los siguientes:

- Rechazo y desconfianza ante las estimaciones. Una estimación debería ser la expresión más honrada que puediera realizarse sobre los recursos y el tiempo necesarios para obtener el producto deseado, así como sobre sus costos asociados. Pero una estimación va a traer frecuentemente noticias desagradables y por ello llegan a buscarse, conscientemente o no, toda clase de excusas para no realizarla.

  Para que la estimación cumpla sus objetivos debe existir - la confianza de que quien la recibe la va a utilizar ade-cuadamente, conociendo sus dificultades y posibles inexactitudes. Si ésto no es así, es muy posible que las estimaciones, si llegan a efectuarse, no tengan ninguna relación con la realidad.

- Es un hecho que los informáticos actúan de forma optimista al enfrentarse con una estimación. En este aspecto es interesante la siguiente cita del artículo "THE MYTHICAL MAN-MONTH", publicado en "Datamation" en Dic.-74 por el Dr. - Brooks, responsable del desarrollo del OS-360 de IBM.

  En este punto conviene anotar, que en muchas ocasiones los defectos más importantes no se dan por la infravaloración-de actividades, sino por su total olvido.

- Presiones sobre quienes estiman. En determinadas ocasiones, por ejemplo cuando se va a utilizar la estimación para establecer un presupuesto o cuando un proyecto comienza a sobrepasarse, es común que se ejerza una presión, a veces indirecta o inconsciente, pero no por ello pequeña, sobre los encargados de estimar, quienes, impulsados además por los otros factores de desconfianza e inexperiencia optimista, y sometidos a las dificultades intrínsecas que se han visto en otros puntos anteriores, producen predicciones necesariamente bajas.

Debe distinguirse claramente entre estimación y compromiso. Este último puede establecerse luego de una estimación y en función de sus resultados, e implicará probablemente la adopción de una serie de acciones. La estimación para ser viable, debe ser lo más libre posible. Si decimos al estimador lo que queremos oir, es muy posible que se sienta tentado de respondernos como un eco. Este es uno de los aspectos que explica más claramente la causa por la que las estimaciones son, con mucha mayor probabilidad, bajas que altas.

TECNICAS DE ESTIMACION

Hasta este momento todas las técnicas de estimación que se utilizan para predecir los costos informáticos se basan en la experiencia, aunque en algunos casos se ha intentado buscar razones técnicas que soporten las expresiones utilizadas.

Existen dos enfoques diferentes (y complementarios) de una estimación:

- De abaja a arriba, o microestimación: el proyecto de que se trata se divide en unidades suficientemente pequeñas (por ejemplo, programas) que se estiman por separado, sumándose posteriormente para dar lugar a la estimación global.

- De arriba a abajo o macroestimación: Se trata de estimar el proyecto en su conjunto, bajando posteriormente a la estimación de sus diversas fases repartiendo la estimación global en función de determinados criterios.

En cuento a la base de la estimación, se dan diversos procedimientos:

- La experiencia propia es una de las técnicas más baratas y empleadas. Consiste en pedir la estimación a alguien que se supone tiene experiencia en casos parecidos, a quien va a realizar el trabajo, etc. Pueden obtenerse mejores resultados, aumentando su costo, consiguiendo la opinión de varios expertos y tratando estadísticamente sus respuestas.

- La comparación con proyectos "parecidos" de los que se disponga de datos. Esta técnica trata de extrapolar al caso actual una experiencia anterior y presupone que se sigue una disciplina de recogida de datos y características de los proyectos que se desarrollan.

- La preparación de estándares, invariantes (dentro de ciertos límites) para las instalaciones, tales como el número de instrucciones programadas y puestas a punto por día y persona, el número de compilaciones necesarias para poner a punto 1.000 líneas fuente, al número de milisegundos de CPU entre dos I/O, el número de sentencias fuente mantenidas por persona, etc. Esta técnica exige recoger datos en cantidades significativas y tratarlos estadísticamente.

LA ESTIMACION DEL DESARROLLO

Es la que más estudios ha producido hasta el momento, probablemente porque en esta fase se dan las desviaciones más llamativas. Más adelante se dan algunos ejemplos de técnicas de estimación que se orientan principalmente a ella, por lo que no se entra aquí en más detalle.

EJEMPLOS DE TECNICAS DE ESTIMACION

A continuación se incluyen, como ejemplos, algunas de las técnicas que se han publicado sobre la estimación de los costos informáticos.

Se ha procurado presentar una panorámica sobre los diferentes enfoques, consideraciones y resultados que permite el tema, clasificándose además, en orden temporal, con lo que se espera que el lector compruebe cómo van evolucionando las -- ideas sobre la estimación.

METODO DE ESTIMACION DE TIEMPO DE PROGRAMACION

El método (desarrollado antes de 1969) pretende predecir los días que son necesarios para realizar un programa. Las - funciones comprendidas son:

- Diseño de la lógica del programa. (No se comprende el diseño de registros, informes, tablas y ficheros, ni la descripción de sus campos, pues se supone realizado en el análisis anterior).
- Codificación.
- Preparación del juego de ensayo.
- Pruebas.
- Documentación.

La estimación se realiza bajo la fórmula:

$$E = C \times (A + B)$$

E: Esfuerzo en días-hombre.
C: Complejidad del programa.
A: Factor de competencia en la programación.
B: Factor del conocimiento del trabajo.

La complejidad del programa se calcula mediante las siguientes tablas de complejidad de entradas/salidas y de procesos. En la primera deben de sumarse todos los puntos que resulten, mientras que en la segunda se entra por lenguaje y -

por complejidad de cada una de las 5 funciones generales que tiene un programa, según este método, sumándose también los puntos obtenidos.

COMPLEJIDAD DE ENTRADAS Y SALIDAS

ENTRADAS	PUNTOS
Tarjetas. Formato único	1
Varios formatos	2
Cada fichero secuencial de cinta o disco	
Un tipo de registro	1
Varios tipos de registro	2
Registros de longitud variable.	3
Cada fichero index-secuencial. Registro único	3
Varios tipos de registro	4
SALIDAS	
Informes. Por cada línea (cabeceras incluídas)	1
Tarjetas. Formato único	1
Varios formatos	2
Cada fichero secuencial de cinta o disco	
Un tipo de registro	1
Varios tipos de registro	2
Registros de longitud variable	3
Cada fichero index-secuencial. Registro único	2
Varios tipos de registro	3
Cada fichero directo. Registro único	3
Varios tipos de registro	4

COMPLEJIDAD DE PROCESOS

		PUNTOS		
				MUY
LENGUAJE	FUNCION	SIMPLE	COMPLEJA	COMPLEJA
COBOL	1. Reestructuración de datos	1	3	4
	2. Condicionamiento	1	4	7
	3. Búsqueda y presentación de los datos	2	5	8
	4. Cálculos	1	3	5
	5. Encadenamiento	1	2	3
	TOTAL	6	17	27
ASSEMBLER	1. Reestructuración de datos	4	5	6
	2. Condicionamiento	4	7	9
	3. Búsqueda y presentación de los datos	4	7	9
	4. Cálculos	3	5	8
	5. Encadenamiento	2	3	5
		17	27	37
UTILITARIOS (SORT, etc.)	Sin "cwn coding"	1	-	-
	Con "own coding"	2	3	4
		3	3	4
RPG		2	8	13

La competencia del programador se da en la siguiente tabla:

EXPERIENCIA	A
Programador muy experimentado	0,5 a 0,75
Experiencia normal	1 a 1,5
Experiencia baja	2 a 3
Programador en formación	3,5 a 4

El factor de conocimiento del trabajo se determina con la siguiente tabla, en función del disponible y del requerido para realizar el programa:

	B		
	CONOCIMIENTO REQUERIDO		
CONOCIMIENTO DISPONIBLE	MUCHO	ALGUNO	NINGUNO
Completo y detallado	0,75	0,25	0
Buen conocimiento general y algo detallado	1,25	0,50	0
Conocimiento general	1,50	0,75	0
Conocimiento de temas relacionados	1,75	1,00	0,25
Ningún conocimiento	2,00	1,25	0,25

METODO DE SVEN ERIKSEN DE ESTIMACION DE PROYECTOS

El método se refiere a proyectos informáticos de gestión. Es muy sencillo y se dirige a comerciales o técnicos que se responsabilizan de instalaciones realizadas para sus clientes. Los datos numéricos que aporta se fundamentan en la experiencia extraída de muchos casos, pero no cita la base estadística. Sus principales características son las siguiente:

- Supone que un proyecto se descompone en tres fases:

  - Análisis del sistema.
  - Diseño del sistema.
  - Implementación del sistema.

  La salida del diseño del sistema es un conjunto de especificaciones de programas y la fase de implementación llega hasta la prueba de los programas no incluye la puesta en marcha.

- La estimación del tiempo (dedicación necesaria) del proyecto se basa en là de la fase de implementación, que se supone ocupa el 40% del tiempo total. El resto (60%) se ocupa en las fases de análisis y diseño, sin especificar los porcentajes respectivos.

- El tiempo necesario para la fase de implementación se obtiene dividiendo el sistema en programas. En el caso de que haya que estimar antes de haberse terminado las fases de análisis y diseño, aconseja dividir el sistema en "componentes lógicos" suficientemente pequeños como para ser - considerados como programas, de acuerdo con la experiencia.

- Divide los programas en dos clases: "normales" y "especiales" (SORTS, utilitarios y otros programas paramétricos).

- La implementación de un programa "normal" consiste en las- siguientes actividades:

  1. Estudio de las especificaciones del programa y establecimiento de su lógica principal.
  2. Establecimiento de la lógica detallada (organigrama) -
  3. Codificación.
  4. Establecimiento del juego de ensayo.
  5. Pruebas.
  6. Documentación del programa.

- El tiempo para implementar un programa "normal", en cada - una de estas actividades, se supone función de la "complejidad", el "tamaño" y el "lenguaje".

- La complejidad viene dada por la siguiente tabla:

  A: Muy fácil.
  B: Fácil.
  C: Moderada.
  D: Difícil.
  E: Muy difícil.

No da otros criterios para establecerla, salvo indicar que no tiene que relacionarse necesariamente con su tamaño. (Un programa grande puede ser muy fácil).

- El tamaño de un programa se define en función del número de líneas de codificación, según la siguiente tabla:

1 :	1	a	400	líneas.
2 :	400	a	600	"
3 :	600	a	800	"
4 :	800	a	1.000	"
5 :	1.000	a	1.200	"
6 :	1.200	a	1.400	"
7 :	1.400	a	1.600	"

- En lenguaje, se distinguen tres clases: Ensamblador básico, Macro-ensamblador y alto nivel (COBOL, FORTRAN, ALGOL).

- En base a la experiencia, propone unas tablas que dan los tiempos dedicados a cada una de las actividades 1 a 6 de la implementación de un programa, así como el tiempo de máquina necesario. Por su interés, se da a continuación un extracto para los programas codificados en lenguajes de alto nivel y la suma de las 6 actividades de implementación.

		DIFICULTAD									
		A		B		C		D		E	
		DH	HM	DH	HM	DH	HM	DH	HM	DH	HM
TAMAÑO	1	14	3	27	4	37	5	48	7	63	9
	2	18	4	31	5	41	6	52	8	67	10
	3	22	5	35	6	45	7	56	9	71	11
	4	26	6	39	7	49	8	60	10	75	12
	5	30	7	43	8	53	9	64	11	79	13
	6	34	8	47	9	57	10	68	12	83	14
	7	38	9	51	10	61	11	72	13	87	15

El cuadro da el número de días-hombre (DH) y horas-máquina (HM) necesarios para implementar los programas de la - dificultad y el tamaño dados en las entradas a la tabla.- De estos tiempos totales es significativo que el 50% de - los días-hombre aproximadamente, se dedican a la preparación del juego de ensayo y a las pruebas, según Eriksen.

Para pasar de estos tiempos a los del proyecto completo - deben añadirse los tiempos de "implementación" de los programas especiales (4 días por cada uno) y el tiempo dedicado a las fases de análisis-diseño (un 150% de los tiempos de implementación).

Como se deduce de estos datos, Eriksen obtiene productividades para proyectos completos del orden de 7 líneas de codificación por día-Hombre para programas medios en dificultad y -

longitud (900 líneas), aunque esta productividad oscila ampliamente desde 2 líneas/día para programas cortos y complejos hasta 16 líneas/día para programas largos y simples.

METODO DE MYERS

Myers (IBM) desarrolló en 1972 un método de estimación de proyectos, aplicable principalmente para casos medios a grandes (típicamente, con equipos de más de 15 personas y 18 a 30 meses de plazo).

El método pasa por las siguientes etapas:

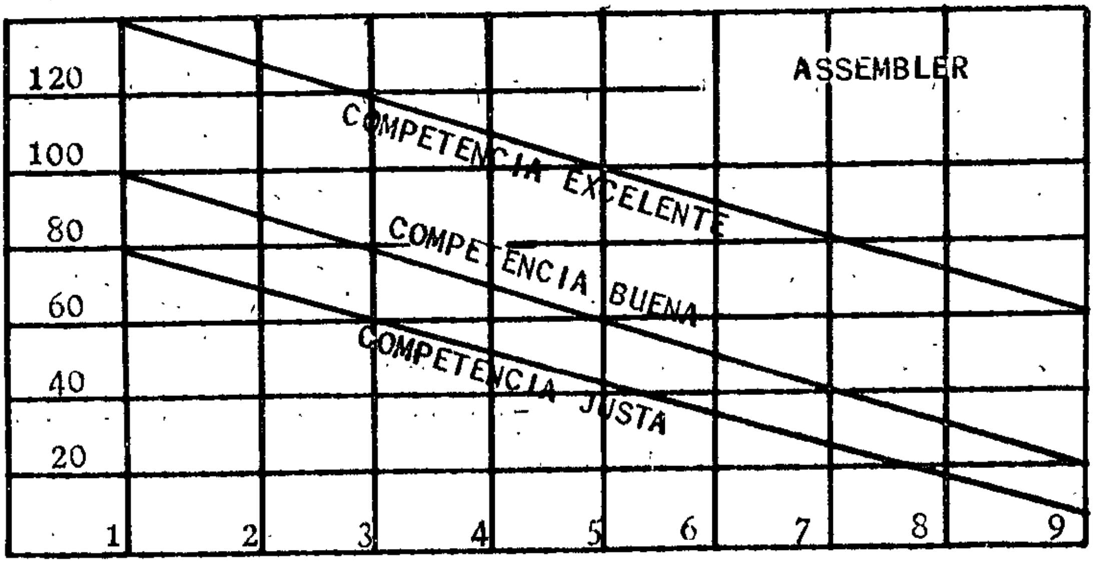
a) Cálculo del número estimado de líneas fuente que comprende el sistema. Este número se estima en tres hipótesis: pesimista (La), más probable (Lm), y optimista (Lb), en función de las opiniones de expertos.

b) Estimación de la productividad durante la fase de programación. Igual que en la etapa anterior, se buscan tres hipótesis de productividad: pesimista (Pa), más probable (Pm), y optimista (Pb). Esto se hace de la manera siguiente:

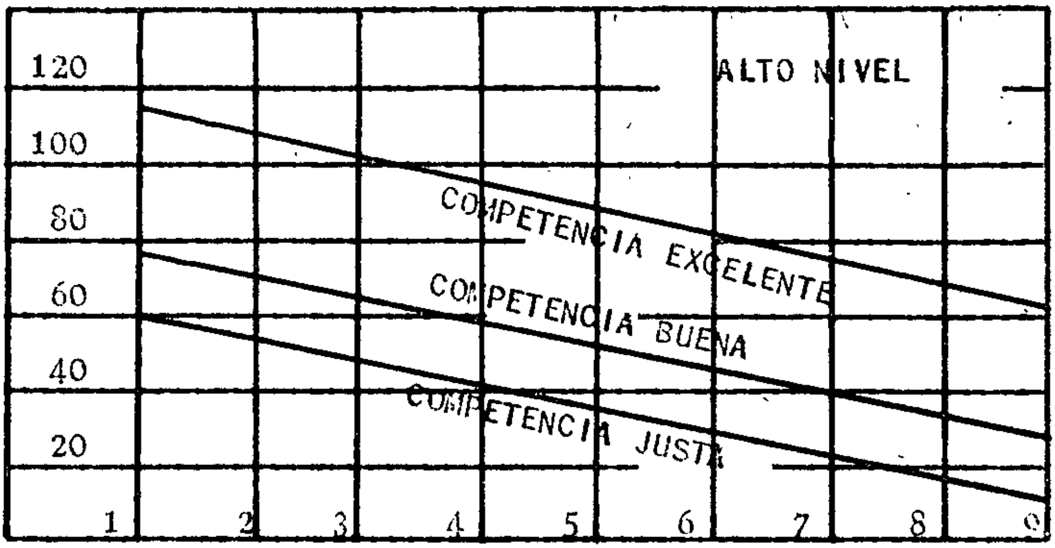
- Asignación de un parámetro de dificultad:
  1: Sistema "fácil" : Pocas interacciones. Entrada/Salida simple.
  2: Sistema "medio" : Algunas interacciones entre los elementos del sistema. Interfases y entradas/salidas medias.
  3: Sistema "difícil" : Programas de tiempo real o con muchas interacciones entre sus diversas partes. Entradas/Salidas complejas.

- Asignación de un parámetro de tamaño al sistema.
  1; Menos de 5.000 líneas.
  2: Entre 5.000 y 25.000 líneas.
  3: Más de 25.000 líneas.

- La multiplicación de ambos parámetros da un "índice de complejidad" que va de 1 a 9, con el que se entra en unas tablas que dan la productividad, en líneas/día, en función de la competencia del equipo de desarrollo (excelente, buena o justa) y del tipo de lenguaje (ensamblador o alto nivel). Estas tablas se dan en la figura adjunta.



INDICE DE COMPLEJIDAD



c) Cálculo del esfuerzo de programación.

Myers propone que se calcule el número de días-hombre necesarios para programación en las tres hipótesis: pesimista (a), más probable (m) y optimista (b), como : EPa= La/Pa, EPm = Lm/Pm, EPb = Lb/Pb, y que se añada a esta necesidad técnica otra de soporte (secretaría, bibliotecario, transcripción, operación, dirección, administración de base de datos, comunicación y técnica de sistemas etc.) calculada también en tres hipótesis, como ESa = Ca. Lm/Pm, ESm = Cm. Lm/Pm, ESb = Cb. Lm/Pm, donde C es un factor que indica la cantidad de soporte necesario, y que se toma según la siguiente tabla, en función del tamaño del sistema:

TAMAÑO	Ca	Cm	Cb
1	1	0,75	0,5
2	2	1,5	1
3	3	2	1,5

d) Esfuerzo necesario para el proyecto.

Según Myers, el esfuerzo de programación es el 50% del total en el proyecto, y, por tanto, en las tres hipótesis anteriores.

Esfuerzo pesimista,	$Ea = 2.$	$(EPa + ESa)$
Esfuerzo más probable,	$Em = 2.$	$(EPm + ESm)$
Esfuerzo optimista,	$Eb = 2.$	$(EPb + ESb)$

e) Esfuerzo en función del riesgo.

En los casos "normales", debería de ocurrir que aproximadamente, $Em = Eb + \frac{1}{4}\ (Ea - Eb)$

Si así sucede, la siguiente tabla puede utilizarse para deducir un esfuerzo E para el cual el riesgo de sobrepesamiento R sea inferior a un cierto porcentaje.

$\frac{E - Eb}{Ea - Eb}$	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8
R(%)	90	70	50	30	20	10	5	2

Supongamos, como ejemplo del método, un sistema cuya dimensión oscilara entre 80.000 y 160.000 líneas, con un valor más probable de 90.000 líneas, considerando como de dificultad "media" y a desarrollarse por un equipo de "buena" competencia en lenguaje de alto nivel.

Su "índice de complejidad" sería 3 x 2 = 6. Según las tablas los tres valores de la productividad de programación esperada resultan Pa = Pm = 43 líneas/día y los tres valores del parámetro C de soporte serían Ca = 3, Cm = 2, Cb = 1,5.

Así se obtienen los siguientes esfuerzos esperados:

Pesimista : Ea = 2 (3.720 + 6.280) = 20.000 jornadas
Más probable: Em = 2 (2.093 + 4.186) = 12.558
Optimista : Eb = 2 (1.860 + 3.140) = 10.000

Puede observarse que $Eb + \frac{1}{4} (Ea - Eb) = 12.500 \simeq 12.558$

Estamos, en consecuencia, ante un caso "normal", y el esfuerzo, según Myers, para un riesgo del 10%, sería ,

E = Eb + 0,6 (Ea - Eb) = 16.000 jornadas = 1.000 meses-hombre (1 mes-hombre = 16 jornadas)

La comparación de este resultado con otros métodos (ver el de Putnam), puede realizarse teniendo en cuenta que al contarse en Myers todo el soporte (incluso administrativo) el costo del mes-hombre debería de calcularse sin repercutir sobre él ningún costo de personal "indirecto".

ESTADISTICAS DE WALSTON Y FELIX

En 1977, estos dos autores publicaron en IBM System Journal el resultado de un estudio estadístico que había comenzado en 1972, cuyo objetivo era estudiar la productividad del desarrollo informático y los factores que le afectan. El estudio, realizado en la división federal de sistemas IBM, se refiere a proyectos de desarrollo y de mantenimiento.

En el momento de su publicación, la base de datos construída con los datos obtenidos, incluía 60 proyectos de desarrollo acabados, cuyas dimensiones iban desde 1.000 hasta 712.000 líneas fuente de codificación, con unos esfuerzos desde 3 meses-hombre hasta 11.758. Estos proyectos correspondían a sistemas muy variados, escritos en más de 20 lenguajes y empleando más de 60 tipos de ordenadores.

En cuanto a los proyectos de mantenimiento, en el estudio no se cita su base estadística y se indica que incluían no sólo detección y corrección de errores, sino también pequeños desarrollos por cambios funcionales.

Los más importantes resultados de este estudio estadístico se dan en los puntos que siguen:

a) Datos referentes a los proyectos acabados

	Mediana	Cuartiles	
	50%	25%	75%
Productividad			
Líneas fuente por mes hombre total	274	150	440
Producto			
Líneas fuente (miles)	20	10	59
Páginas de documentación por mil líneas fuentes	69	27	167
Recursos			
Costo ordenador (% sobre total)	18	10	34
Esfuerzo total meses-hombre	67	37	186
Equipo medio	6	3,8	14,5
Distribución del esfuerzo (%)			
- Dirección, Administración	18	12	20
- Análisis	18	6	27
- Diseño y programación	60	50	70
- Otros	4	0	6
Duración (meses)	11	8	19

b) Datos referentes a los proyectos de mantenimiento acabados

	Mediana	Cuartiles	
	50%	25%	75%
Producto			
Líneas fuente mantenidas (miles)	103	56	474
% líneas nuevas desarrolladas durante el servicio	4	0	19
Recursos			
Equipo medio	6	4	11
Líneas mantenidas por hombre (miles)	15	5	24
Distribución del esfuerzo			
- Dirección, Administración	15	10	20
- Análisis	10	4	30
- Programación	72	40	80
- Otros	3	0	7
Duración (meses)	18	11	31
Errores detectados			
Por mil líneas mantenidas	1,4	0,2	2,9

c) Ajustes de los principales parámetros

Walston y Felix publican en su trabajo diversos gráficos y expresiones que relacionan entre sí los parámetros más importantes. El procedimiento general para su obtención ha sido el ajuste de funciones exponenciales a las nubes de puntos experimentales.

Las principales fórmulas son las siguientes:

- Esfuerzo (E, meses-hombre) en función de la dimensión -- (L, miles de líneas fuente)

$$E = \begin{Bmatrix} 12,7 \\ 5,2 \\ 2,1 \end{Bmatrix} L^{0,91}$$

La llave indica que el coeficiente vale 5,2 en media, habiendo un 68% de probabilidades de que se encuentre entre 12,7 y 2,1.

Debe tenerse en cuenta que las líneas fuente incluyen aquí las de lenguaje de control (JCL), las definiciones de datos, los comentarios, etc., pero no las reutilizadas (COPY, etc.).

También se indica que el esfuerzo total incluye la dirección, la administración, el análisis, el soporte de explotación, la documentación, el diseño, la codificación y las pruebas.

- Productividad: La anterior expresión indica una relación - casi lineal entre el esfuerzo y el tamaño del sistema, aunque un amplio márgen de variación en la productividad. Para reducirlo y hacer estimaciones más ajustadas, los autores buscaron la correlación entre la productividad y otros factores, encontrando valores significativos en 29 de los

Variables	Productividad media			Incremento de productividad
	-1	0	+1	P
Complejidad de relación con el usuario	Normal 124	Normal 295	Normal 500	376
Participación del usuario en los requerimientos	Mucha 201	Alguna 267	Ninguna 491	286
Experiencia y cualificación generales del personal del proyecto	Baja 132	Media 257	Alta 410	278
Experiencia previa en aplicación de similar o mayor tamaño y complejidad	Mínima 146	Media 221	Extensa 410	264
Experiencia previa en el lenguaje de programación	Mínima 122	Media 225	Extensa 385	263
Porcentaje de personal de desarrollo que participó en el análisis funcional	25% 153	25,50% 242	50% 391	238
Restricciones referentes a memoria en los programas	Severas 193	Medias 277	Mínimas 391	198
Utilización de la organización "Programador Jefe"	0-33% 219	33-66% ---	66% 408	189
Complejidad del proceso de la aplicación	Media 168	Media 345	Media 349	181
Experiencia previa en el ordenador de explotación	Mínima 146	Media 270	Extensa 312	166
Número de clases de items en la base de datos por cada 1.000 líneas	80 193	16-80 243	0-15 334	141
Acceso cerrado al ordenador	85% 170	11-85% 251	0-10% 303	133
Secreto en cuanto al ordenador y el 25% de los programas	Si 156		No 289	133
Ratio del tamaño medio del equipo (personas) a la duración (meses)	0,9 173	0,5-0,9 310	0,5 305	132

	Productividad media			Incremento de productividad
Variables	-1	0	+1	P
Utilización de programación estructurada	0-33% 169	34-66% Sin datos	66% 301	132
Restricciones de tiempo sobre los programas	Severas 171	Medias 317	Mínimas 303	132
Acceso al ordenador abierto bajo demanda especial	0 226	1-25% 274	25% 357	131
Complejidad general de los programas	Media 185		Media 314	129
Desarrollo de "arriba a abajo"	0-33% 196	34-66% 237	66% 321	125
Páginas de documentación, por 1.000 líneas fuente	88% 195	33-88 252	0-32 320	125
Desarrollo simultáneo de hardware y software	Si 177		No 297	120
Inspecciones en el diseño y la programación	0-33% 220	34-66% 300	66% 339	119
Experiencia del usuario en la aplicación	Mucha 206	Alguna 340	Ninguna 318	112
Restricciones generales sobre los programas	Severas 166	Medias 286	Mínimas 293	107
Porcentajes de líneas fuente finalmente entregadas	90% 159	91-99% 327	100% 265	106
Cambios originados por el usuario	Muchos 196		Pocos 297	101
Complejidad de los organigramas	Media 209	Media 299	Media 289	80
Programas sin cálculos complejos	0-33% 188	34-66% 311	67-100% 267	79
Programas para tiempo real o para ejecutarse bajo restricciones severas de tiempo	40% 203	10-40% 337	10% 279	76

68 que fueron estudiados. Estos 29 factores, así como las variaciones en la productividad a la que dan lugar, se detallan en los cuadros adjuntos.

A partir de estos datos de define el "Indice de productividad" para un proyecto mediante la expresión:

$$I = \sum_{1 \text{ a } 29} Wi\ Xi$$

Wi (peso de la variable i) = $\frac{1}{2} \log_{10} \Delta Pi$

Xi (valor de la variable) = +1, 0, -1 según la respuesta en el proyecto corresponde a la primera, la segunda o la tercera columna de los cuadros citados.

Calculando este Índice en los proyectos a los que se refiere el estudio, los autores encuentran una correlación significativa con la productividad (en líneas por mes hombre), y un ajuste que, según los gráficos publicados, responde a la fórmula:

$$P = \left\{ \begin{matrix} 469 \\ 274 \\ 160 \end{matrix} \right\} 10^{0,029\ I}$$

Debe advertirse que en el estudio no se consideran las posibles correlaciones entre diversas variables.

- Documentación (D, páginas), en función a la dimensión -

$$D = \left\{\begin{matrix} 135 \\ 49 \\ 18 \end{matrix}\right\} L^{1,01}$$

La documentación considerada incluye los listados de los programas.

- Duración (M, meses) en función de la dimensión o del esfuerzo.

$$M = \left\{\begin{matrix} 7,3 \\ 4,1 \\ 2,3 \end{matrix}\right\} L^{0,36}$$

$$M = \left\{\begin{matrix} 3,6 \\ 2,5 \\ 1,7 \end{matrix}\right\} E^{0,35}$$

- Equipo medio (S, personas) en función del esfuerzo total

$$S = \left\{\begin{matrix} 0,83 \\ 0,54 \\ 0,35 \end{matrix}\right\} E^{0,6}$$

- Costo de ordenador (C, en miles de dólares) en función de la dimensión o del esfuerzo total

$$C = \left\{\begin{matrix} 4,44 \\ 1,84 \\ 0,76 \end{matrix}\right\} L^{0,96}$$

$$C = \left\{ \begin{matrix} 2,2 \\ 1,1 \\ 0,5 \end{matrix} \right\} E^{0,81}$$

d) Ejemplo.

Según estas fórmulas, puede estimarse que un proyecto de - 100.000 líneas con un índice de productividad 0 precisaría de un esfuerzo que el 68% de las veces estaría entre 625 y 213 meses-hombre, siendo el valor más probable el de 365 - meses-hombre. Este proyecto generaría 5.000 páginas de documentación, duraría 20 meses, con un equipo medio de 18 - hombres y el costo de ordenador (precios del 72 al 77) sería de 125.000 $. Todos estos valores tienen, según este- estudio, fuertes márgenes de variación.

METODO DE PUTNAM

Putnam, presidente de una empresa de servicios especializados en estimaciones de costo de software, ha desarrollado - en 1979, un método de estimación aplicable a grandes proyectos de sistemas fuertemente enterrelacionados (sistemas "integrados", sistemas operativos, etc.) que típicamente tienen - más de 100.000 líneas de codificación.

El método se basa en dos hipótesis obtenidas de la experiencia en grandes proyectos.

a) Los recursos humanos que precisa un proyecto responden a - la siguiente ecuación (curva de Rayleigh)

$$s = \frac{E}{T^2} t e - {}^{t^2 / 2T^2}$$

s: personas del equipo en el momento t.

E: Esfuerzo total necesario para el desarrollo y el mantenimiento del sistema, en años-hombre.

T: Tiempo de desarrollo del sistema, en años.

Algunas propiedades de esta curva son las siguientes:

- El pico de personas necesario se da en el momento T en que acaba el desarrollo, y es igual a 0,61 E/T.

- El esfuerzo consumido hasta el momento t (integral de -- la curva anterior) responde a la fórmula:

$$E (1 - e^{t^2 / 2T^2})$$

Según ella, al finalizar el desarrollo se ha consumido el 39% del esfuerzo total que se empleará en todo el ciclo de vida del proyecto (queda, por lo tanto, un 61%, que se destina al mantenimiento.

Otros puntos de interés en estas expresiones son:

Punto	t / T	% Esfuerzo
Certificación del diseño	0,43	9
Prueba del sistema integrado	0,67	20
Prueba del prototipo	0,80	27
Comienzo de la instalación	0,93	35
Capacidad operativa completa	1,00	39

b) La dimensión del sistema, el esfuerzo necesario para desarrollarlo y el plazo se relacionan entre sí según la siguiente expresión.

$$L = A \cdot E^{1/3} T^{4/3}$$

L: Número de líneas fuente (en miles) de que se compone finalmente el sistema.

A: Constante que depende de la metodología utilizada.
Se propone un valor de 10.

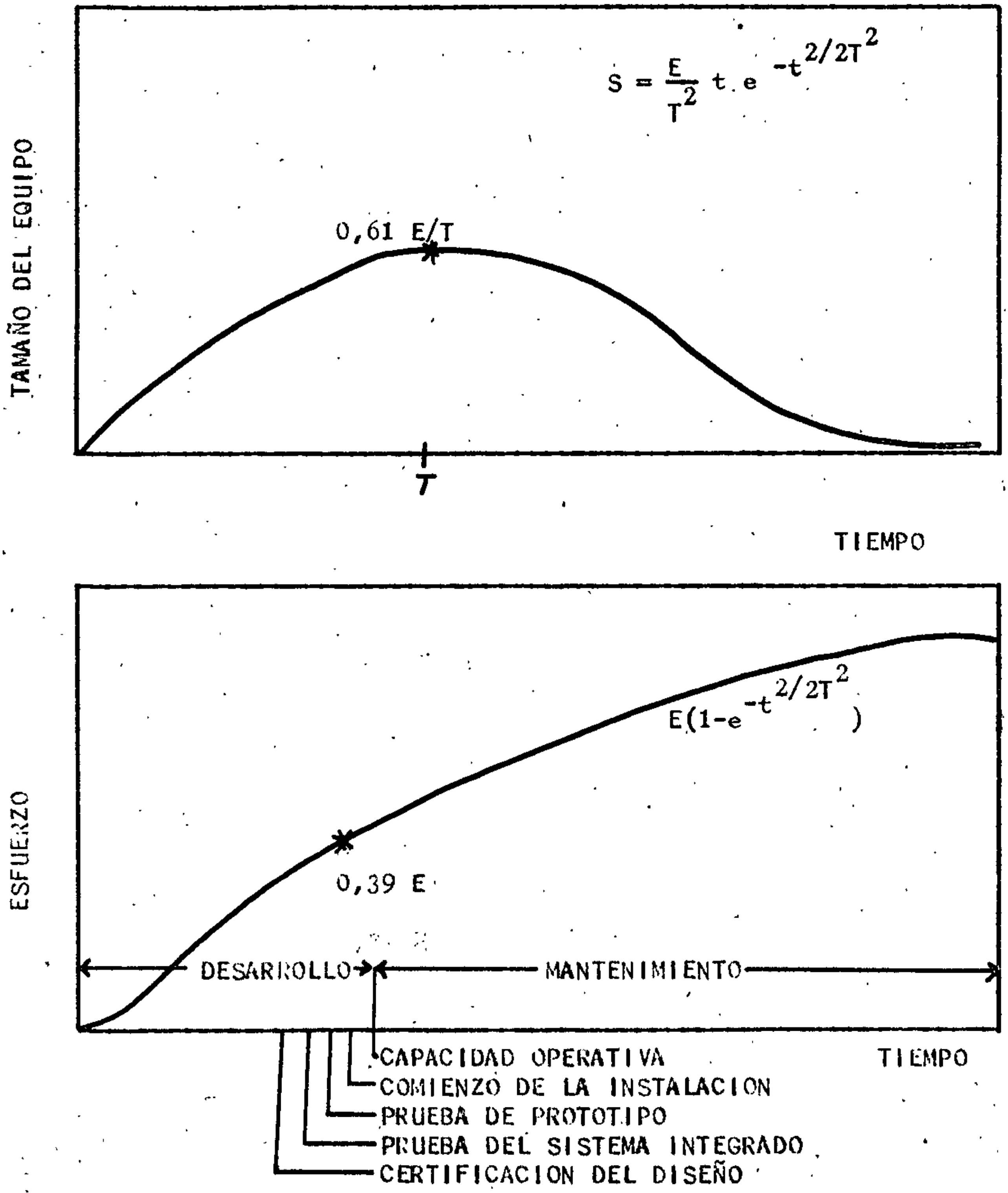
E, T: Esfuerzo total (años-hombre) y tiempo de desarrollo (años) según se definieron antes.

De esta ecuación pueden deducirse otras más significativas

Meses de desarrollo: $M = 2{,}29\ L^{3/5} .S^{-1/5}$

Costo de desarrollo: $C = 0{,}21.K.L^{3/5} .S^{4/5} = 0{,}091.K.M.S.$

Productividad (líneas/día-h) $P = 15{,}7\ (M/S)^{2/3}$


$S = \frac{E}{T^2} t \cdot e^{-t^2/2T^2}$
TAMAÑO DEL EQUIPO
0,61 E/T
T
TIEMPO
ESFUERZO
$E(1-e^{-t^2/2T^2})$
0,39 E
DESARROLLO
MANTENIMIENTO
TIEMPO
CAPACIDAD OPERATIVA
COMIENZO DE LA INSTALACION
PRUEBA DE PROTOTIPO
PRUEBA DEL SISTEMA INTEGRADO
CERTIFICACION DEL DISEÑO

S: Personal medio durante el desarrollo (=0,39 E/T)

K: Coste (incluyendo cargas indirectas, máquina que consume etc.) de un hombre-año.

En estas fórmulas se supone que 1 año-hombre se compone de 11 meses-hombre o de 176 días-hombre de trabajo efectivo - para el proyecto.

Mediante estas dos hipótesis pueden estimarse facilmente - los principales parámetros de los grandes proyectos, para- lo cual, además, Putnam sugiere:

- Estimar la dimensión de los sitemas en consultas con expertos, obteniendo las sentencias fuente mínimas (La), - más probables (Lm) y máximas (Lb) que se juzguen razonablez, a partir de las que se calculan:

$$L = \text{Dimensión estimada} = \frac{La + 4Lm + Lb}{6}$$

$$\sigma = \text{Desv. típica} = \frac{Lb - La}{6}$$

suponiendo ahora una distribución normal, se espera no - sobrepasar:

L con un 50% de probabilidad
$L + \sigma$ " " 84% " "
$L + \sigma 2$ " " 97% " "
$L + \sigma 3$ " " 99% " "

- Aplicar la fórmula anterior a varias situaciones de equipo de trabajo y dimensión de proyecto, a fin de dar a la dirección del proyecto una idea sobre los riesgos existentes.

- Realizar este mismo proceso varias veces a lo largo de la vida del proyecto, para obtener resultados cada vez más seguros.

Como ejemplo, supóngase que se trata de un sistema cuya dimensión mínima, al finalizar la fase de especificación, se supone de 80.000 líneas, la más probable de 90.000 y la máxima de 160.000.

El tamaño estimado del sistema será:

$$L = \frac{80 + 4 \times 90 + 160}{6} = 100$$

$$\sigma = \frac{160 - 80}{6} = 13$$

Así que los tamaños y sus riesgos serán:

Tamaño menor de	Riesgos de sobrepasamiento
100	50%
113	16%
126	3%

Aplicando las fórmulas anteriores, y suponiendo un costo total por hombre-año de 4 Mptas., se obtiene la siguiente tabla en varias hipótesis sobre el equipo de trabajo (P) y riesgo de sobrepasamiento (R).

MESES DESARROLLO				COSTO DESARROLLO (Millones Ptas.)			PRODUCTIVIDAD Líneas/día.		
S/R:	50%	15%	3%	50%	16%	3%	50%	16%	3%
5	26	28	30	48	52	55	48	50	52
10	23	25	26	84	90	96	27	28	30
15	21	23	24	116	125	133	20	21	22
20	20	21	23	146	157	168	16	17	17
25	19	21	22	174	188	201	13	13	14
30	18	20	21	202	218	232	12	12	12

Si se desease acabar el desarrollo en 2 años (22 meses) con un riesgo inferior al 3%, habría que disponer de un equipo medio de 25 personas (es decir, un pico de 39 personas) y el proyecto costaría 201 MPts. Ahora bien, si se admite un riesgo de sobrepasamiento de este plazo del 16%, el equipo podría disminuirse hasta 20 personas, con lo que su costo bajaría hasta 157 MPts.

En todos estos casos, una multiplicación de los costos obtenidos por 1,5 daría la cifra que, según este método, se estima debería dedicarse a la corrección de errores y a las modificaciones del sistema posteriores a su implantación (mantenimiento.

Como puede observarse en este ejemplo, el método de Putnam indica una fuerte incidencia de los plazos de desarrollo en los costos. Ello es debido a que sus fórmulas tienen en cuenta el hecho de que en los grandes equipos de trabajo se emplea buena parte del tiempo en intercomunicación entre sus miembros, bajando de forma importante la productividad al aumentar el tamaño del equipo (derivaciones de la "ley de Brooks").

La llamada "ciencia del software" representa una nueva - oriéntación en el aspecto de medida del software, y se encuentra actualmente en un perfodo de desarrollo y experimentación. Sus bases fueron publicadas por Halstead en 1977..Su aplicación a la estimación de costos está aún lejana, aunque parece posible.

Sus principios son los siguientes:

1. En un programa es posible e interesante medir:

   n_1 : El número de diferentes operadores usados. Un operador es un código de operación, un sfmbolo aritmético, un sfmbolo de puntuación, un paréntesis, etc.

   n_2 : El número de diferentes operandos usados. Un operando es una constante o una variable.

   N_1 : El número total de ocurrencias de los operadores de- un programa.

   N_2 : El número total de ocurrencias de los operandos de - un programa.

   Se llama VOCABULARIO n de un programa a la suma $n_1 + n_2$.

   Se llama LONGITUD N de un programa a $N_1 + N_2$.

2. Estadfsticamente se comprueba que la longitud de un programa puede estimarse a partir de su vocabulario mediante la siguiente ecuación:

$\hat{N}$ estimada = $n_1 \log_2 n_1 + n_2 \log_2 n_2$

Esta relación se da con más exactitud cuando los programas están mejor estructurados y cuando se han eliminado algunas impurezas tales como las siguientes: operandos sinónimos, operandos utilizados para más de una variable, reaplazamientos innecesarios, operadores que se cancelan entre si, etc.

3. Otras definiciones de interés son:

- Volumen del programa: $V = N\log_2 n$.
  Representa el número de bits necesario para especificar un programa (elegir para cada uno de sus elementos un componente de su vocabulario).

- Nivel del programa: Puesto que un programa puede escribirse en forma más o menos abstracta, el "nivel" trata de medir la relación entre su volumen en la forma más abstracta (compacta) posible y su volumen real.
  Un estimador para este nivel es $L = \frac{2}{n_1} \cdot \frac{n_2}{N_2}$

- Nivel del lenguaje: Coincide con el significado que normalmente se da a este concepto. Se estima, a partir de los programas, como $1 = L^2 V$.

4. Estimaciones deducidas de la ciencia de software:

- Esfuerzo mental requerido para crear (o quizás entender) un programa: Es directamente proporcional a su volúmen e inversamente a su nivel, es decir: $e = V / L$.

- Tiempo de programación: Debe ser proporcional al esfuerzo mental. Según Halstead, $E = e / c$, donde c es una constante de proporcionalidad, que parece próxima a 18 para una medida de E en segundos-hombre.

  Representa el número de discriminaciones que la mente humana es capaz de realizar por segundo. Se supone una persona concentrada y sin ninguna dificultad en el manejo del lenguaje.

- Errores cometidos durante la programación (antes de realizar ninguna compilación ni prueba). Pueden estimarse como $B = e^{2/3} / 3.200$.

  Esto equivale a considerar que en cada 3.200 decisiones binarias tomadas por la mente humana, una de ellas es errónea o, puesto que se toman hasta 18 decisiones/segundo, se comete un error cada 3 minutos a la máxima concentración.

CONCLUSIONES

De igual manera que sucede en otras actividades, la estimación de costos es necesaria en informática para gestionar adecuadamente este recurso. Por una parte facilita datos para tomar decisiones sobre la oportunidad de aplicaciones y modificaciones, sobre las alternativas de adquisición o realización, sobre los posibles diseños, etc. y por otra es imprescindible para planificar y controlar eficazmente los proyectos de desarrollo.

Las técnicas de estimación se basan hasta ahora siempre en la experiencia y tienen importantes componentes subjetivos, no siendo posible, en general, conseguir estimaciones muy seguras. Sin embargo, la aplicación sistemática de una técnica puede conducir a resultados mucho más fiables que los que se obtuviesen por un procedimiento intuitivo.

Las técnicas publicadas parten, en general, de una estimación subjetiva del tamaño (líneas fuente) del sistema a desarrollar, y de algunos otros parámetros (dificultad, plazo de entrega, etc.) y dan lugar a estimaciones de trabajo (meses-hombre) mediante el empleo de tablas o expresiones matemáticas.

No hay ninguna técnica que se adapte, sin más, a una instalación concreta. Si se desea implantar una, deben realizarse algunos estudios estadísticos sobre el pasado y recoger sistemáticamente datos de los proyectos que se vayan realizando y de los técnicos que intervienen en ellos.

ORGANIZACION DE RECURSOS

Esta es una etapa muy importante en el desarrollo de proyectos. Es aquí donde se realiza la distribución o asignación de - los recursos disponibles con el fin de minimizar el costo y evitar duplicidad de funciones.

Existen un gran número de estilos y maneras de organizar un proyecto. A continuación se describirán tres tipos de organización, de las cuales se han llevado a cabo más frecuentemente por los Directores de Proyectos:

- Organización funcional
- Organización por proyecto
- Organizaciones matriciales y mixtas.

ORGANIZACION FUNCIONAL:

Esta organización se basa en la especialización del trabajo, con un responsable de cada grupo de especialistas. En los departamentos de informática, estas especialidades pueden ser: Análisis funcional, Análisis orgánico, Programación, Mantenimiento, -- etc. Esta forma de organización tiene la ventaja de posibilitar un buen ajuste y homogeneidad en las metodologías, pero, para los proyectos, tiene el defecto de diluir la responsabilidad, ya que, cuando se da en forma pura, implica que, en cada momento, haya - partes de cada proyecto distribuidas en varios grupos, cuyos criterios sobre prioridades pueden no ser coherentes. El único responsable de cada proyecto resulta ser, en consecuencia, el director del departamento.

ORGANIZACION POR PROYECTOS.

Este tipo de organización, en su forma pura, consiste en establecer en el departamento de informática un grupo de trabajo,- con un responsable, para cada proyecto o conjunto de proyectos - existente, grupo del que dependerían jerárquicamente todos los - especialistas necesarios.

ORGANIZACIONES MATRICIALES Y MIXTAS.

Tratan de aprovechar las ventajas de las dos organizaciones extremas vistas hasta aquí. En ellas existen por una parte grupos funcionales, con sus responsables, y por otra directores de proyectos, cargos que, en algunos casos, son temporales y pueden nombrarse entre personal tanto de informática como de usuarios. Dentro de estas organizaciones se encuentran muchas variedades. Algunas típicas son las siguientes:

- Funcional coordinada: El personal pertenece jerárquicamente a grupos funcionales pero está asignado por tiempos limitados a los directores de proyecto, que los piden y los devuelven según sus necesidades y de acuerdo con el director del departamento o de algún comité de asignación de personal.

- De proyecto coordinada, en la que el personal depende jerárquicamente de los responsables de proyectos, existiendo un staff técnico que se preocupa de coordinar la metodología y detectar situaciones anormales en la ocupación del personal de cada grupo.

- Mixta matricial, en la que una parte del personal pertenece jerárquicamente a grupos funcionales y otra parte está permanentemente integrada en pequeños grupos de proyecto, que toman, - de los grupos funcionales y por tiempo limitado, el resto del personal especialista que precisan.

Todas estas organizaciones matriciales tienen como principal problema el que determinadas personas están dependiendo durante cierto tiempo de dos jefes, lo cual puede provocar rechazos por inestabilidad y falta de coordinación.

En consecuencia, quedan, como características de la organización de proyectos, las siguientes condiciones:

- Hay un DIRECTOR DE PROYECTO, el cual tiene unas responsabilidades muy amplias sobre el proyecto que de él depende, tanto en su gestión como su liderazgo técnico.

- Este director tiene asignados unos recursos, que se concretan en un EQUIPO DE TRABAJO cuya composición y organización son - generalmente variables durante la realización del proyecto, - según las necesidades de cada momento. Los miembros de este- equipo, según los casos, pueden depender jerárquica o sólo -- funcionalmente del director de proyecto, pueden estar asignados permanentemente o temporalmente al equipo y pueden proceder de informática, del usuario o del exterior, pero, en cualquier caso, durante su asignación al grupo, su trabajo es planificado y controlado por el director del proyecto o sus mandos intermedios.

Esta programación se hace, en general, al comienzo de cada fase, y se va corrigiendo periódicamente (por ejemplo, cada mes).

LANZAMIENTO: El final de la actividad de planificación consiste en la asignación de trabajos, ya definidos y evaluados, a - subgrupos o a técnicos concretos, según su disponibilidad y capacidad, con quienes se establece un compromiso sobre los objetivos a cumplir (plazos, presupuesto, calidad, etc.).

El director del proyecto debe establecer, en cada momento, la asignación más adecuada, creando a veces subgrupos temporales (dirigidos por mandos intermedios), estableciendo comités de seguimiento y coordinación, designando especialistas para labores de consultoría interna, etc.

Este seguimiento puede reflejarse periódicamente (por ejemplo, anualmente), en informes dirigidos a los jefes jerárquicos del personal que forma parte temporalmente de su equipo.

LA COORDINACION: Es otra de las responsabilidades del director del proyecto, quien debe mantener coordinados, informados y animados a los diversos subgrupos y a los miembros de su equipo.

Además, y para el personal que depende jerárquicamente de él,- deberá realizar todas las actividades propias de esta dependencia (permisos, salarios, ascensos, amonestaciones, etc.).

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

DOCUMENTACION DE PROYECTOS EN INFORMATICA

DEFINICION DE REQUERIMIENTOS DEL USUARIO

(DOCUMENTACION BASICA)

FEBRERO, 1984

alacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40 20 Apdo. Postal M 2285

V. DEFINICION DE REQUERIMIENTOS DEL USUARIO (DOCUMENTACION BASICA)

1.- Introducción

1.1. ¿Qué es?.-

La fase de "definición de requerimientos" es la siguiente fase despues de la de "planeación" en el "ciclo de vida de los sistemas", se encarga de definir los objetivos en detalle del sistema (planteamiento de objetivos), coleccionar información para posteriormente ser analizada llegándose a plantear y seleccionar alternativas (análisis), y finalmente especificar de una manera formal y objetiva las cualidades deseadas en el producto final así como el medio ambiente en el cual el usuario lo operará (especificación de requerimientos).

En esta fase se especificará el qué va a realizar el sistema no el cómo lo realizará, siendo esto parte de la fase de diseño. Lo que de alguna manera es ambiguo o mal entendido será traducido a una especificación concreta que es la base para la fase del diseño.

Las tareas que se realizarán estarán enfocadas en la descripción de la información de las funciones y de la ejecución del sistema. Cada característica del sistema será refinada por un analista que actuará junto con el usuario y por otro lado con el diseñador. Estas características serán modeladas por medio de un conjunto de herramientas como diagramas de flujos de datos para el flujo de información, diagramas jerárquicos o diagramas warnier-orr para la estructuración de información o para un enfoque automatizado como puede ser el PSL/PSA.

Como documento final en esta fase se tiene el de especificación de requerimientos.

1.2. Problemática.-

En la etapa de definición de requerimientos pueden encontrarse una serie de problemas los cuales traen como consecuencia pérdidas de tiempo y esfuerzos, incremento de costos e insatisfacción de las necesidades del usuario entre otras. Los problemas más comunes en esta etapa son:

A. Problemas de comunicación.

B. Cambios en la especificación de requerimientos

C. Falta de conocimiento

D. El humano como procesador de información

E. Parcialidad en la selección y uso de datos

F. Duplicación de esfuerzos

G. Carencia de herramientas

A. Problemas de comunicación.-

El principal problema para la determinación de los requerimientos es la mala comunicación entre el usuario y el analista creando un medio en el cual es difícil alcanzar el objetivo principal: la determinación de los requerimientos del usuario.

Aunque los requerimientos del usuario son la base para todas las futuras decisiones y actividades, si son incompletos o inadecuados se garantizará un deficiente producto fi nal.

Esta mala comunicación se da generalmente por los diferentes lenguajes que emplean las partes involucradas pudiéndose solucionar por medio de un lenguaje común a ambas partes. También se da por los métodos inapropiados de representar ideas.

B. Cambios en la especificación de requerimientos.-

En algunas ocasiones los requerimientos del usuario son modificados ya sea por mal entendimiento o por adiciones al sistema. Esto trae como consecuencia utilizar recursos adicionales para la actualización y/o retrasos del proyecto. Considerando que los requerimientos son difíciles de "congelar" (no hacerles modificaciones) es necesario que el documento de "especificación de requerimientos" sea altamente actualizable y mantenible para reducir al máximo el impacto en los cambios.

C. Falta de conocimiento.-

Los usuarios en algunas ocasiones aún no conocen lo suficiente acerca de los sistemas de procesamiento de información para saber qué es factible y qué no. Dada la gran propaganda que se le hace a las computadoras la gente no tiene las ideas adecuadas para saber qué puede o no hacer la computadora. Por esta razón es recomendable sensibilizar al usuario de las bondades y limitaciones de las computadoras, en esta tarea el analista tendrá una gran ingeren cia.

D. El humano como procesador de información.-

El ser humano utiliza tres tipos de memorias en el procesamiento de información:

+ memoria externa
+ memoria de término largo y
+ memoria de término corto

La memoria externa está constituída de medios externos por medio de los cuales se puede guardar información, tal es el caso de un libro de una nota de remisión o de un dispositivo de despliegue visual (pantalla).

La memoria de término largo pertenece al cerebro humano teniendo una capacidad esencialmente ilimitada. Se requiere sólo unos pocos centenares de milisegundos para leer de ella, pero por el contrario para escribir (depositar) en ella se emplea bastante tiempo.

La memoria de término corto es una memoria muy rápida pero de pequeña capacidad. Haciendo una analogía con la computadora vendría siendo un registro o memoria cache. La memoria de término corto es utilizada para el procesamiento de información, para operaciones de cálculo y de comparación. Estas limitaciones afectan la habilidad humana para definir requerimientos.

La capacidad de una memoria de término corto está caracterizada como un "siete más o menos dos". El 7 ± 2 se refiere a la colección de datos. La colección de datos puede estar dentro del rango de un caracter hasta una imagen visual. Así, por ejemplo, un número telefónico puede llenar la memoria de término corto cuando es marcado para realizar una llamada, o las imágenes de siete personas pueden ser almacenadas durante el procesamiento humano de seleccionar una persona.

Los límites de la memoria de término corto afectan los requerimientos de información obtenidos, siempre y cuando el proceso utilizado para la obtención de los requerimientos utilice sólo la memoria de término corto (esto es sin ayuda de memoria externa).

El usuario entrevistado no podrá retener un gran número de elementos en la memoria de término corto para propósitos de discusión y análisis, y es por esto que se limitan las respuestas en esta tarea.

También la limitación de este tipo de memoria puede afectar el número de requerimientos que el usuario defina como importantes. En varias actividades de la entrevista utilizando la memoria de término corto, el usuario puede haber seleccionado algunos elementos de información y registrar éstos en la memoria de término largo como los más importantes. Esta información puede ser sólo recordada cuando se le hace una pregunta sobre ella.

Las limitaciones de memoria de término corto pueden ser significantemente reducidas si se utiliza memoria externa para almacenar datos que serán procesados y por la utilización de metodologías que sistemáticamente produzcan y registren pequeños números de cantidades de datos.

E. Parcialidad en la selección y uso de datos.-

Existe una evidencia substancial que muestra que los humanos no son imparciales en la selección y uso de los datos. Los efectos en la determinación de los requerimientos de información son significativamente parciales hacia los requerimientos basados en los procedimientos utilizados en ese momento (actuales), su información disponible, etc. Tanto el analista como el usuario que conocen estas parcialidades pueden compensarlas; un método significativo de compensación es la de proporcionar una estructura para la solución del problema.

F. Duplicación de Esfuerzos.-

Cuando el documento de "especificación de requerimientos" es escrito de tal manera que sea enfocado sólo a los usuarios, este no podrá ser muy útil a los diseñadores físicos y programadores que construirán el sistema. Esto trae como consecuencia que se realice un re-análisis, duplicando de esta manera los esfuerzos de tal manera que exista una redefinición de datos y procesos en términos que el diseña--

dor puedan entenderlo fácilmente.

G. Carencia de herramientas.-

En ocasiones el analista sólo utiliza su conocimiento práctico junto con su lápiz y papel para determinar las necesidades del usuario sin utilizar herramientas prácticas, tanto manuales (diagramas de flujo de datos, árboles de decisión, diagramas warnier, etc.) como automáticas (procesadores de textos, diccionarios de datos, etc.) para la determinación de dichas necesidades.

1.3. Relaciones Usuario/Analista.-

La relación usuario/analista es fundamental para la obtención de los requerimientos del usuario, los papeles que juegan se describen a continuación:

a) Usuario.- Existen 3 tipos de usuarios:

El operador del sistema quien se encargará de manejar el sistema para la explotación de la información.

El responsable, quien es el responsable directo de los procedimientos que van a ser automatizados y que fundamental-- mente proporcionará los requerimientos al analista.

El dueño del sistema, quien es el que paga el sistema o es un administrador de alto nivel.

b) Analista.- El analista es la principal liga entre el área de los usuarios y el área que realizará el sistema. Realiza o coordina cada una de las tareas asociadas en la fase de definición de requerimientos. Cuando recolecta información se comunica con el usuario para averiguar las características del medio ambiente existente y sus necesidades.

El analista deberá tener las siguientes habilidades:

+ Habilidad en el dominio de conceptos abstractos, reorganizarlos dentro de divisiones lógicas y sintetizar soluciones basado en cada una de las divisiones.

+ Habilidad para comprender el medio ambiente del usuario.

+ Habilidad para aplicar los elementos de software y/o hardware a las necesidades de los usuarios.

+ Habilidad para comunicarse de manera verbal y escrita con facilidad.

+ Habilidad para detectar la información confusa o conflictiva.

Dado que el analista generalmente no está familiarizado con las actividades del usuario, depende de la información que el usuario quiera proporcionarle. Para que la información sea completa debe tenerse una adecuada cooperación por parte del usuario, la que se alcanza con tacto, psicología y sentido común. El primer contacto que se realiza con el usuario es de gran importancia para las actividades futuras. Por lo tanto es necesario tener buenas relaciones desde un principio.

En las relaciones entre el usuario y el analista deben de solucionarse los problemas de comunicación, con lo que se combatirían los siguientes puntos de vista:

+ El usuario cambia los requerimientos del sistema.
+ El usuario no comunica sus necesidades con la precisión deseada.
+ El usuario pocas veces tiene tiempo para comunicarse con el analista.
+ El usuario no entiende lo que se le explica.
+ El analista toma mucho tiempo para realizar un objetivo.
+ El sistema cuesta más de lo calculado.
+ El sistema no funciona como se esperaba.
+ El analista no entiende las necesidades del usuario.
+ El analista no se comunica en un lenguaje adecuado.

1.4 ¿Qué debe proporcionar el usuario?.-

Dado que dentro de las actividades del analista se encuentra la de recolectar información para ser analizada, se da a continuación una lista que deberá ser proporcionada por el usuario:

+ Trámites.- Se refiere básicamente a la solicitud de sistematización.
+ Organización.- Se refiere a la estructura orgánica por parte del área del usuario para, de esta manera, iden-

tificar las relaciones organizacionales que guardan los usuarios, así como el flujo de información que existe en la organización.

+ Definición del Problema.- Definir los problemas que serán solucionados por medio de la sistematización.

+ Procedimientos.- Qué procedimientos se siguen en el proceso a sistematizar.

+ Entradas.- Se refiere a la definición de entradas: cuáles son, qué volumen, qué frecuencia.

+ Salidas.- Se refiere a la definición de salidas: cuáles son, qué volumen, que frecuencia.

+ Prioridades.- Cuáles son las necesidades a corto, mediano y largo plazo.

+ Seguridad.- Seguridad en la información para accesarla.

+ Controles.- Qué controles se aplicarán a la información en lo que se refiere a los datos de entrada y salida, así como a los aspectos operativos.

+ Parámetros de Prueba.- Colección de información que servirá para verificar si el producto final satisface los requerimientos del usuario.

La Administración

La administración de proyectos es un proceso que se encuentra presente a lo largo de todo el proyecto, es por esto que la administración también está involucrada en la fase de "Definición de Requerimientos". La administración está involucrada en todas las actividades que se relacionen con el cuidado y realización del proyecto. Dicha administración se llevará a cabo por el administrador del proyecto o por el líder del proyecto.

Como en todo proyecto es necesario contar con un plan actualizado antes de iniciar una fase o una etapa, donde se identifiquen las actividades a realizar, los programas de tiempo y la estimación de los costos, todo esto con el objeto de transformar las metas de la fase o etapa en actividades de trabajo pormenorizadas que, al combinarse con las restricciones de tiempo y recursos, originen los productos finales ya definidos con el mínimo costo. Es oportuno considerar que el plan propuesto deberá ser aprobado por la gerencia para continuar con el proyecto y, de esta manera además, comunicarles sobre su camino y costos. Este plan será almacenado en el archivo del proyecto el cual contendrá adicionalmente los documentos y minutas relativos a la administración.

Se identificarán los productos finales que deberán generarse en la fase, con el objeto básico de servir como un checklist de las tareas asignadas a los miembros del equipo del proyecto. Estos productos podrán variar en cada instalación, encontrándose básicamente: los del análisis y los de especificación de requerimientos.

Se organizará la asignación de responsabilidades a las diferentes áreas involucradas con el fin de no repetir esfuerzos.

Se organizará la asignación de las actividades a los diferentes miembros del equipo de trabajo y el monitoreo de los avances del proyecto, además se registrarán las fechas y costo de las actividades terminadas. El monitoreo del avance de la fase servirá para verificar si el trabajo está realizándose de acuerdo a lo prescrito, en caso de existir desviaciones éstas deberán ser justificadas para poder ser ajustadas con previa autorización de la gerencia.

Se elaborará un plan de entrevistas con el usuario con la finalidad de obtener información necesaria para el adecuado desenvolvimiento del proyecto.

Se elaborará un plan preliminar de pruebas para conocer qué tipos de pruebas se realizarán.

Se realizará un calendario de revisiones informales durante la fase contemplando además las personas involucradas.

Mediante revisiones controlar los documentos generados en la fase para su aprobación.

La información servirá para actualizar el archivo del proyecto, el cual consistirá en los reportes de trabajo, los planes propuestos, las minutas de las entrevistas y las revisiones, entre otros.

2.- Análisis de Necesidades

La etapa de análisis tiene como objetivos realizar una recolección de datos, ordenarlos de una manera útil y analizar su integridad y exactitud, además de analizar las características del sistema de una manera sistemática y comprensible.

2.1 Recolección de Datos.-

La tarea de recolección de datos es muy importante ya que la integridad y exactitud de los datos gobiernan la precisión del análisis. Existen tres técnicas básicas de recolección de datos. Dos de ellas dependen en gran medida de la participación de los usuarios involucrados. La otra produce los datos menos utilizables. Es aconsejable mezclar estas tres técnicas para asegurar una mayor integridad de los datos. Las tres técnicas de recolección disponibles para el analista son:

Revisión de Documentos.- Esta técnica tiende a ser considerada por los analistas más valiosa de lo que realmente es, quizás por que en muchos departamentos las cosas que "realmente" son importantes son puestas por escrito. Por lo general los analistas examinan las descripciones del trabajo y los procedimientos formales escritos con el fin de comprender las funciones y responsabilidades del usuario. Muchos analistas no se dan cuenta de que la información dentro de las áreas del usuario regularmente no es mejor que la que se podría tener internamente en el área del analista. En resumen, a menudo no refleja el estado actual de las operaciones.

La revisión de documentos puede proporcionar un cuadro global de una área funcional, una buena estructura histórica, e información sobre las condiciones básicas y la razón fundamental para las políticas y procedimientos que se llevan en ese momento. Es sin embargo, una fuente de información muy insegura sobre los métodos y procedimientos utilizados en ese momento porque la documentación formal es actualizada unicamente cuando existe suficiente tiempo -lo cual no sucede a menudo.

Entrevistas.- Dado que las entrevistas son la fuente principal de datos en la mayoría de los proyectos de sistemas, deberán ser conducidas para maximizar la transferencia de información entre el analista y el usuario. Es de suma importancia que las relaciones usuario/analista sean óptimas.

El analista debe proyectar una actitud de objetividad, neutralidad y diplomacia, el actuar de esta manera puede tener un efecto positivo sobre la calidad de los datos recibidos, ya que existirá gran cordialidad y cooperación en la entrevista. La apariencia de objetividad, neutralidad y diplomacia es enormemente díficil de lograr a menos que sea genuina, aunque puede lograrse a través de las experiencias obtenidas. El analista de menor experiencia, sin embargo, puede beneficiarse del asesoramiento e instrucción proporcionados por un supervisor, sobre la importancia de estos tres aspectos.

El propósito de la entrevista inicial con cada usuario es el de obtener tanta información utilizable como sea posible. Esto se consigue mejor estructurando la entrevista con pregun--

tas que fuercen al usuario a dar respuestas descriptivas. Preguntas tales como: ¿Qué realiza usted?, ¿Qué cosas le gustaría ver cambiadas?, ¿Qué problemas tiene?, etc., resultan generalmente en un flujo de información que proporciona estructura y detalle sin la potencial parcialidad de las preguntas específicas.

Es importante señalar que durante la entrevista el analista deberá evitar hacer juicios sobre el valor o impresión de los datos recibidos al usuario, principalmente porque el usuario puede malentender las cosas y por consiguiente no ser cooperativo en la siguiente entrevista, despues de un análisis inicial de las respuestas, para la clarificación y expansión de los datos específicos.

El analista en la mayoría de las ocasiones va a tener en número y tiempo restringidas las entrevistas con el usuario, por lo que es recomendable que después de analizar la información previa de una entrevista se le hagan preguntas que sirvan para alcanzar la integridad y exactitud deseadas. Las entrevistas con el usuario pueden realizarse por medio de algunas técnicas de preguntas entre las que se encuentran las preguntas cerradas, preguntas abiertas y lluvia de ideas.

Observación Personal.- El tercer método de colección de datos no es utilizado comunmente por los analistas de sistemas, debido a su previa asociación con las funciones de tiempos y movimientos y análisis de procedimientos manuales. La observa ción personal de las operaciones es un método excelente para determinar cómo son realmente ejecutadas las operaciones diarias. La revisión de documentos por un lado indica cómo se supone que funcionan las operaciones o cómo acostumbran funcionar; las entrevistas proporcionan información sobre cómo quiere el usuario que sean presentadas las funciones en ese momento; y la observación, por otro lado, proporciona información acerca de cómo el usuario ejecuta realmente sus funciones en una situación de producción.

Con este método se tienen porblemas ya sea porque el analista de sistemas no está acostumbrado a observar las cosas por períodos largos, o porque el supervisor del analista le llama la atención por no entregar resultados. Sin embargo, es justo mencionar que la observación es el único médio en el cual la información recolectada no llega filtrada al analista. Es por esto que los datos obtenidos de esta manera son inmediatamente aceptados como exactos dentro del contexto de la situación.

Una vez completada la colección inicial de los datos, el analista se enfrenta a una gran cantidad de datos incoherentes de indeterminada calidad y exactitud. El analista debe organizar los datos colectados de manera útil y analizar su integridad y exactitud antes de que los requerimientos del usuario puedan ser formulados.

2.2 Análisis de la Información.-

El análisis de la información es una actividad que sirve para desarrollar una especificación completa, consistente, inambigua y comprensible de la información recolectada que puede servir como base para el común acuerdo de todas las partes involucradas, describiendo qué es lo que el sistema realizará.

Se tienen básicamente dos enfoques en el análisis: el clásico y el estructurado.

El enfoque clásico se basa en la presentación de un documento que describe el propósito del sistema. El documento es poco entendible ya que contiene bastantes tecnicismos en varios cientos de páginas para que el usuario lo revise. En la mayoría de los casos estos usuarios no tienen tiempo ni el conocimiento técnico para entender las implicaciones del sistema propuesto.

Dentro de los problemas del enfoque clašico con respecto al documento que describe el propósito del sistema, se tiene que:

+ Es monolítico. Está compuesto de una sola parte, lo cual dificulta encontrar información dentro de él.

+ Es redundante. La misma información es repetida en varios puntos del documento.

+ Es difícil de modificar y mantener. Para realizar un cambio tiene que modificarse el documento en varios puntos. Consecuentemente es difícil mantener el documento actualizado.

+ Es físico en lugar de lógico. En algunas ocasiones los requerimientos del usuario son descritos en términos de dispositivos físicos o de los tipos de estructura que utilizará el sistema, invadiéndose al cómo el sistema realizará sus funciones.

Todo esto redunda en un análisis que abarca mucho tiempo y que es muy difícil modificar.

El enfoque estructurado se basa en la utilización de herramientas de documentación gráficas para producir una nueva clase de especificación -una especificación estructurada.

Dentro de sus características se tiene que:

+ Es gráfico. Se utilizan diagramas para su representación.

+ Es particionado. Está compuesto de varias "mini-especificaciones", las cuales componen la especificación y que pueden ser tratadas independientemente.

+ Es jerárquico. Se realiza de arriba hacia abajo (top down) llevándose a cabo la tarea desde el nivel más alto y abstracto hasta el nivel más bajo y detallado.

+ Es mantenible. Las modificaciones que se le realicen son fáciles de actualizar.

+ Diferencía las consideraciones físicas y lógicas, y asigna responsabilidades basado en esta diferenciación entre las áreas involucradas.

+ Representa el flujo de información en cualquier sistema:- manual automático o híbrido.

+ Se basa más en el flujo de datos que en el flujo de control.

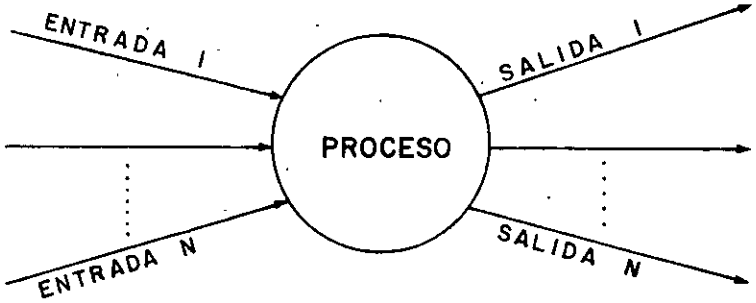
+ Permite al usuario por medio de un modelo del sistema, familiarizarse con las características de dicho sistema antes de implementarse para, de esta manera, poder perfeccionarlo.

El Modelo.-

Una manera de poder lograr la comunicación entre el usuario y el analista es la utilización de un modelo. El modelo es una representación de un sistema del mundo real; el sistema tiene entradas, aplica sus elementos para transformar las entradas en salidas y de esta manera produce las salidas, las cuales al igual que las entradas pueden ser de varias formas. Por ejemplo: las entradas pueden ser señales de control transmitidas por un emisor, una serie de números digitados por alguna persona o un voluminoso archivo de datos recuperado de memoria secundaria. La transformación puede ser una comparación lógica, un algoritmo numérico de gran complejidad o una búsqueda de un elemento. La salida bien puede ser un reporte de 50 páginas o el encendido de un motor.

Un concepto general utilizado por el analista en el desarrollo de los modelos es el uso de un escenario que puede ser una descripción verbal o algunas veces gráfica de todos los eventos en los cuales el usuario tiene experiencia en la realización de las tareas que constituyen la solución del problema del sistema. El escenario es particularmente útil si es preparado con el lenguaje y conceptos que el usuario conoce.

Una técnica para representar un modelo es ilustrada a continuación:



La función total del sistema es representada por una simple transformación de información llamada "burbuja". Una o más entradas mostradas con las flechas son transformadas para producir la información de salida.

Por medio del modelo puede representarse la naturaleza de la información de un sistema, tanto del flujo como de la estructura de la información.

Clases de Metodologías Estructuradas.-

Para la realización del análisis se cuenta con dos tipos de metodologías estructuradas, las cuales utilizan una perspectiva descendente. Dependen básicamente de la descomposición jerárquica.

Las existentes son por flujo de datos o procesos orientados y por estructura de datos.

Análisis estructurado por el método de flujo de datos.-

Estos métodos se basan en la transformación de las funciones más que en la transformación de datos. El énfasis está en encontrar soluciones de diseño desglosando un problema en partes independientes. Estas partes son nuevamente estudiadas y la descomposición repetida. Dentro de las herramientas de flujo de datos puede nombrarse el Diagrama de Flujo de Datos (DFD) y la herramienta automática SADT.

Como la información se mueve a través del sistema; ésta es modificada por una serie de transformaciones. Un Diagrama de Flujo de Datos muestra el flujo de la información y las transformaciones que le son aplicadas, así como el movimiento de los datos de la entrada a la salida.

El análisis estructurado propuesto por Tom DeMarco utiliza el Diagrama de Flujo de Datos como herramienta para su desarrollo.

Análisis estructurado propuesto por Tom DeMarco.-

Como se dijo anteriormente el enfoque estructurado se basa en la utilización de herramientas de documentación gráficas para producir una nueva clase de especificación -una especificación estructurada.

Las herramientas de documentación del análisis estructurado son:

+ Diagrama de Flujo de Datos (DFD)

+ Diccionario de Datos (DD)

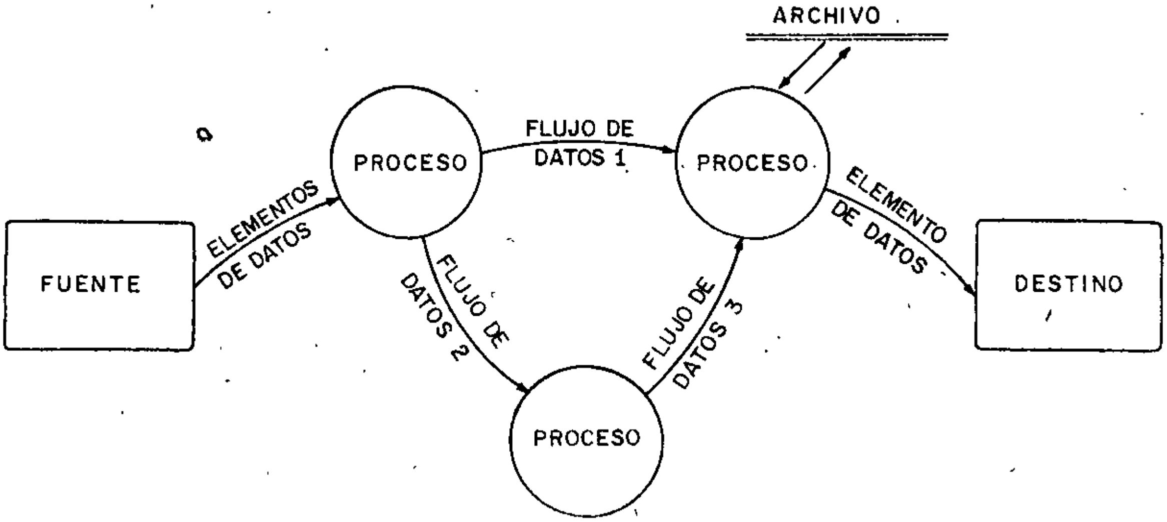
+ Español Estructurado
+ Arboles de Decisión
+ Tablas de Decisión

Diagrama de Flujo de Datos.- El DFD es una herramienta que proporciona un adecuado medio gráfico de modelado del flujo de datos a través de un sistema de cualquier tipo (manual, automático o híbrido). Es una representación de red que retrata al sistema en términos de sus piezas componentes, con todas las interfases entre las componentes involucradas.

Los elementos que componen un DFD son:

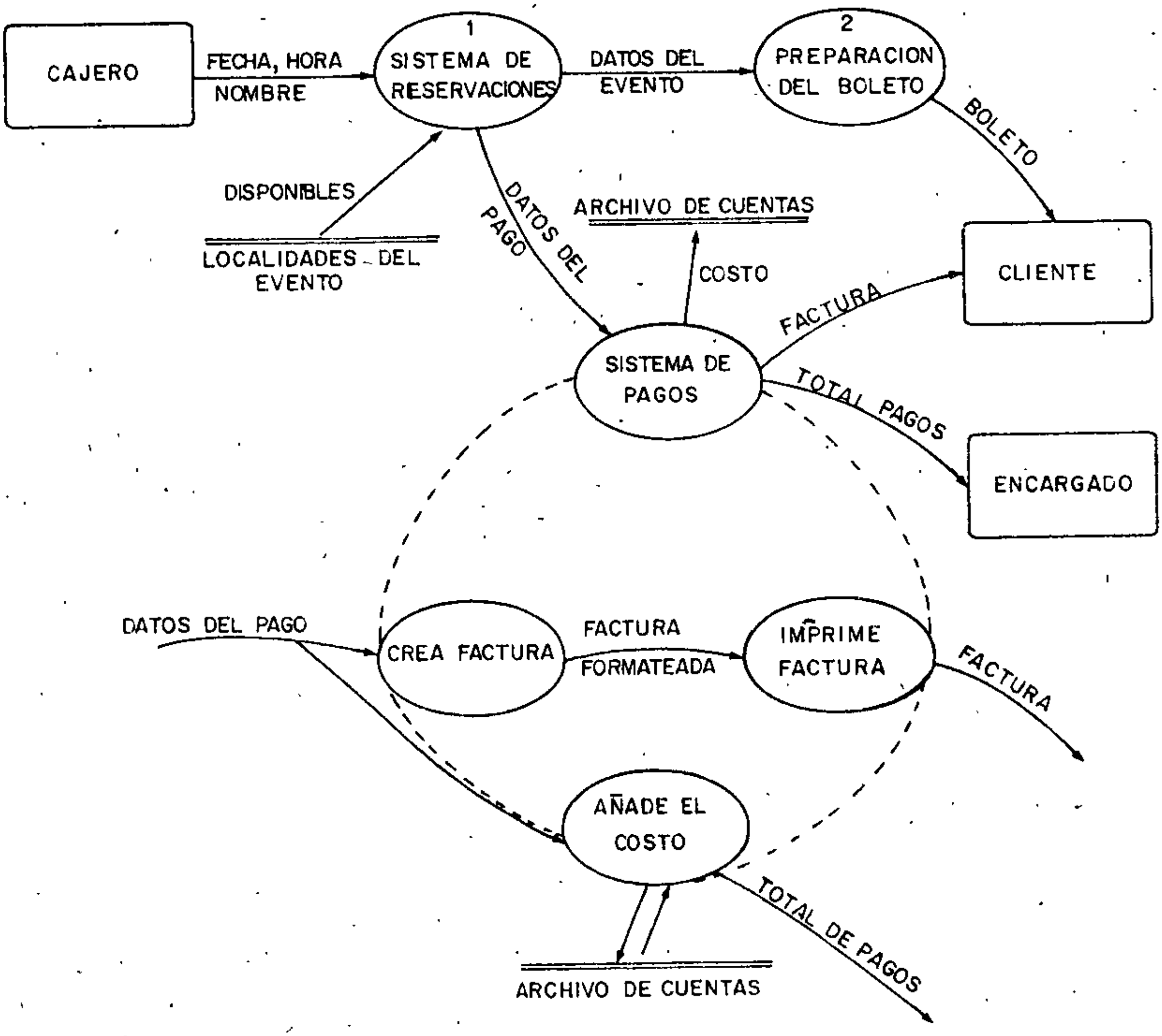
+ Las fuentes y destinos de información representados por las cajas,
+ los flujos de datos (información), representados por las flechas,
+ los procesos (transformaciones), representados por los círculos o burbujas, y
+ los archivos de datos, representados por 2 líneas paralelas verticales,

los cuales se muestran en la siguiente figura:



Una fuente de información es un sitio donde se originan los datos (p.e. una máquina, una persona). Un destino de información es el destino final de un dato que ha sido transformado por el sistema (p.e. un cliente, un usuario).

Por lo general un sistema es lo bastante abstracto que requiere varios niveles de DFD's, por lo que debe subdividirse en nuevos DFD's de manera jerárquica, siendo éstos de mayor detalle. Debe tenerse en cuenta que tanto los flujos de entradas como de salidas no deben perderse, esto es que los flujos deben de ser iguales que los del proceso que se subdividió. Este concepto se muestra en el siguiente ejemplo:



En el ejemplo, un cajero (fuente de información) proporciona información al sistema de venta de boletos, compuesto de tres procesos (burbujas) como se muestra. Los datos que proporciona el cajero son transformados en boletos, facturas y el total de pagos de los boletos para el cliente y el encargado (destinos de información). Los archivos de datos (localidades del evento y cuentas) proporcionan también información en el DFD.

La subdivisión en los DFD debe alcanzar el siguiente detalle en los procesos:

+ tengan sólo una entrada y sólo una salida, o
+ no existan flujos de datos internos por lo que ya no existiría una transformación, o
+ cuando pueden escribirse las especificaciones del proceso en una hoja.

Las fuentes y destinos de información representan entidades exteriores del sistema que se está analizando. Son de donde se originan los datos fuente y los destinos de los resultados producidos.

Los flujos de datos son conjuntos de datos a los que se les asocia un nombre. No pueden tenerse dos flujos con el mismo nombre. Todos los flujos de datos deberán tener un nombre, en caso contrario puede decirse de antemano que existen fallas.

Los procesos representan las transformaciones o funciones que modifican los flujos de entrada en flujos de salida. Cada proceso deberá tener un nombre que identifique la transformación que se le aplicará a los datos de entrada para obtener los datos de salida.

Los archivos es donde se encuentran los datos deben tener un nombre significativo.

Cabe mencionar que los DFD's van a ser dibujados en repetidas ocasiones, ya sea por falta de información o por falta de entendimiento; este trabajo es tedioso pero debe estarse preparado para ello. En la actualidad existe una herramienta automática (H.A.) que facilita en gran medida los cambios en los DFD's y en sus herramientas relacionadas. De dicha H.A. se hablará posteriormente.

Se recomienda que un DFD esté contenido en una sola hoja tamaño carta y tenga 7±2 burbujas para que, de esta manera sea entendible, modificable y fácil de localizar.

Diccionario de Datos.- El Diccionario de Datos (DD) es la segunda herramienta principal en el análisis estructurado. Es simplemente una colección organizada de definiciones lógicas de todos los nombres que están especificados en el DFD. Su objetivo es el de poder entender la información de cada nombre definido en el DFD.

Los nombres que aparecen en el DD deben de ser todos los que aparecen en los diferentes niveles -desde el más abstracto hasta el más elemental del D.F.D. Estos nombres son de:

+ Flujos de datos,
+ Componentes de los flujos de datos,
+ Archivos, y
+ Procesos

Como se observa el DD también incluye los procesos, lo que es puramente lógico, no solo datos; por esta razón su nombre está mal aplicado, realmente debería ser llamado directorio del DFD o directorio del proyecto.

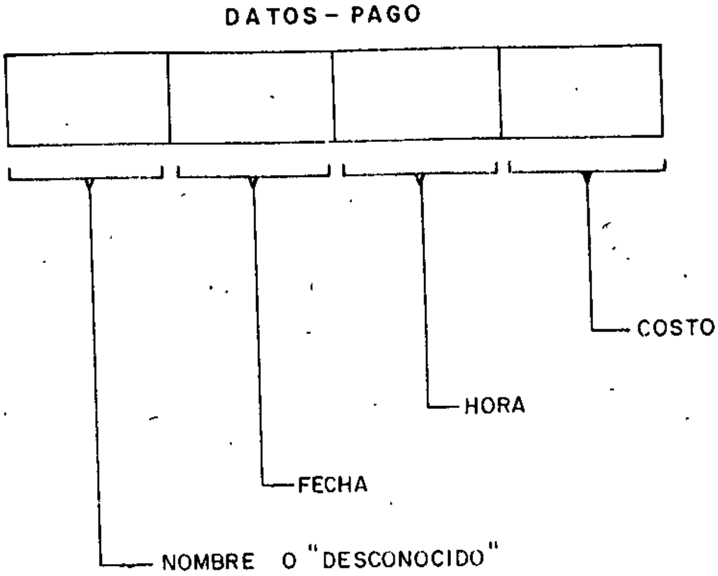
La información que contendrá el DD debe ser objetiva y concisa, evitar la redundancia y que sea de fácil acceso la definición por su nombre. La información más común que se incluye durante el desarrollo del sistema es la siguiente:

+ frecuencia
+ volumen
+ tamaño
+ usuarios afectados
+ consideraciones de seguridad
+ prioridad
+ fecha de implementación
+ definición

En el DFD anterior "datos del pago" podría ser definido en el DD de la siguiente manera:

```
Datos-pago    = (nombre/"desconocido")
                + fecha
                + hora
                + costo
```

que significa que "datos-pago" está compuesto del nombre del que compró o de la cadena "desconocido", de la fecha y de la hora del cuento y del costo del boleto; esto queda representado con la siguiente figura:



Para la descripción de un elemento sólo nos bastará con su nombre y su descripción.

Para la descripción de una estructura de datos, es conveniente utilizar una cierta notación para las estructuras opcionales, repetitivas, etc. La notación puede estar basada en operadores matemáticos para su entendimiento. Estos son definidos de la siguiente manera:

$x=a+b$	x está compuesto del elemento a y b
$x=[a/b]$	x está compuesto de a o b
$x=(a)$	x está compuesto de un elemento a opcional
$x=\{a\}$	x está compuesto de Ø o mas ocurrencias de a
$x=y\{a\}$	x está compuesto de y o más ocurrencias de a
$x=\{a\}z$	x está compuesto de z o menos ocurrencias de a
$x=y\{a\}z$	x está compuesto de y a z ocurrencias de a

Por ejemplo:

Una estructura de datos o un elemento que esté dentro de un paréntesis, significa que es un componente opcional -puede o

no existir- de la estructura:
(nombre)

Dos o más estructuras de datos y/o elementos que estén dentro de paréntesis cuadrados y separados por el símbolo "/" significa que sólo uno de los componentes existirá en la instancia dada de la estructura

[a/b/c]

Una estructura de datos o un elemento podrá ocurrir de y a z ocurrencias con el siguiente formato:

y {estructura} z

La descripción del flujo de datos deberá contener:

+ nombre

+ fuente

+ destino

+ estructura de datos

+ volumen de cada estructura o transacción

La descripción de los archivos deberá contener:

+ nombre

+ flujos de datos de entrada

+ flujos de datos de salida

+ contenido

+ llave de acceso

+ organización física

La descripción de los procesos deberá contener:

+ nombre
+ entradas
+ resumen de la lógica
+ salidas
+ referencia de la especificación funcional

Para el resumen de la lógica se cuenta con varias herramientas entre las que se encuentran los diagramas de decisión, los árboles de decisión y el español estructurado; dichas herramientas serán tratadas posteriormente.

El DD que se implemente puede ser manual, automático o híbrido, y dependiendo del tipo de aplicación se elegirá. Dentro de las ventajas del automático se tienen:

+ Permite la estandarización.
+ Tiene mayor capacidad, soportando de esta manera las cuatro clases de elementos esenciales -flujo de datos, elementos, archivos y procesos.
+ Omite la redundancia.
+ Permite la fácil actualización.
+ Permite varios tipos de reportes (p.e. orden alfabético, referencias cruzadas, etc.).

Español Estructurado.- El español estructurado es el tercer elemento principal del análisis estructurado. Es un subconjunto del lenguaje español compuesto de un limitado conjunto de verbos y sustantivos, organizados para representar un compromiso razonable entre la legibilidad y el rigor. El objetivo del español estructurado es la de permitir al analista describir rigurosa y precisamente las políticas del sistema (sin involucrarse en las tácticas de la implementación), representadas en cada nivel de los procesos del DFD. La descripción debe de ser comprensible para los usuarios.

En el sentido más estricto, el español estructurado consta sólo de los siguientes elementos:

+ Un conjunto limitado de verbos (p.e. encontrar, imprimir, borrar)
+ Estructuras de control permitidas en la programación estructurada (p.e. si-entonces-de lo contrario, haz-mientras, en-caso-de).
+ Elementos definidos en el diccionario de datos.

Con el español narrativo se tienen problemas de ambigüedad, los cuales pueden ser solucionados con el español estructurado. A manera de ejemplos serán descritos.

Problemas con el "no solo", "pero no obstante", "y/o al menos"

a.- "Suma A a B al menos que A sea menor que B, en cuyo caso resta A de B."

b.- "Súmale A a B. Sin embargo, si A es menor que B la respuesta es la diferencia de A y B."

c.- "Súmale A a B, pero resta A de B cuando A es menor de B."

Como se observa, no existe una diferencia lógica entre ellas, pero sin embargo es difícil entender su significado. Cada una puede ser representada por medio de un si-entonces-de lo contrario.

a) acción 1, al menos condición -1, en cuyo caso acción -2

```
Si  cond-1 entonces
    acción-2
de lo contrario (no cond-1)
    acción-1
```

para que en esta instrucción la acción-1 quede en el mismo orden del español narrativo, se niega la condición

```
Si  no cond-1 entonces
    acción-1
de lo contrario  (cond-1)
    acción-2
```

equivalente a

```
Si A no es menor que B entonces
    suma A a B
dè lo contrario
    resta A de B
```

b) acción-1; sin embargo, si cond-1, acción-2

c) acción-1, pero acción-2 cuando cond-1

De lo anterior se desprende que se deben identificar las acciones y condiciones del español narrativo para ser traducidos al español estructurado.

Problemas con "mayor que, menor que".-

La siguiente instrucción: "Menos de 20 artículos comprados no tienen descuento. Más de 20 tienen un 10% de descuento", como se observa, no describe claramente los límites, ya que de 1 a 19 artículos comprados no tienen descuento y de 21 en adelante sí lo tienen. La pregunta es ¿qué sucede cuando se com-- pran 20 artículos?. El problema radica en que el español narrativo no tiene inclusión, esta inclusión se debe realizar con otro conjunto de palabras (p.e. "cantidad menor que o igual a 20").

El español estructurado nos permite verificar todos los posibles rangos con los siguientes términos

Término	Significado
GT	mayor que (greater than)
GE	mayor que o igual a (greater than or equal to)
LE	menor que o igual a (less than or equal to)
LT	menor que (less than)
EQ	igual a (equal to)

Entonces, si también existe descuento para 20 artículos se tendría

```
Si cantidad de artículos GE 20 entonces
    descuento del 10%
de lo contrario
    no existe descuento
```

Problemas de ambiguedad entre el y/o.-

La siguiente instrucción: "los coches que sean negros y que tengan llantas radiales o que sean modelo 83 no pagarán entrada al autocinema"; como se observa, no describe claramente cuándo se pagará la entrada, si cuando son modelo 83 únicamente o cuando son negros y modelo 83. Por medio de paréntesis podremos con el español estructurado separar las condiciones, p.e. si deseamos que cuando el coche sea modelo 83 únicamente, nos quedaría la siguiente instrucción:

```
Si (coche es negro y tiene llantas radiales)
   o (coche es modelo 83) entonces
      no paga entrada
de lo contrario
      sí paga entrada
```

Si por el contrario queremos que no pague entrada cuando es negro y además debe cumplir que tenga llantas radiales o sea modelo 83, nos quedaría la siguiente instrucción:

```
Si (coche es negro) y
   (coche tiene llantas radiales o es modelo 83)
   entonces
      no paga entrada
de lo contrario
      sí paga entrada
```

Problemas en la definición de adjetivos.-

En ocasiones los adjetivos no están bien definidos ya que cada persona entiende las cosas según su conveniencia. Por ejemplo:

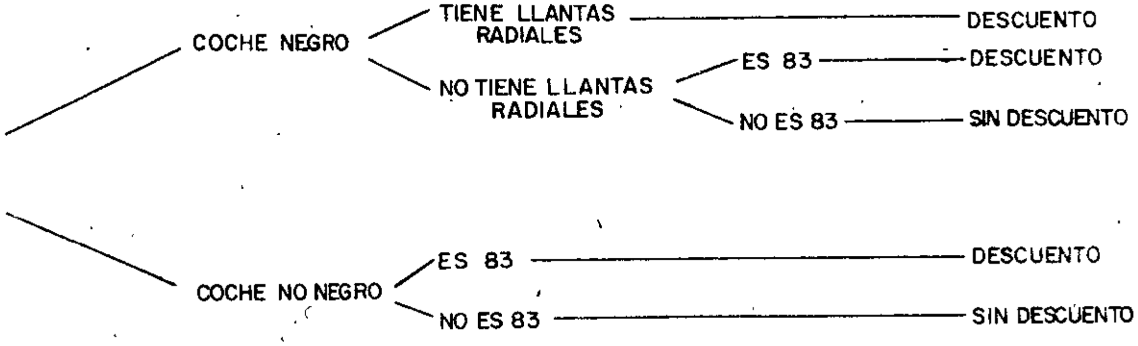
¿Qué es una "completa" cuota?

¿Qué es un "buen" cliente?

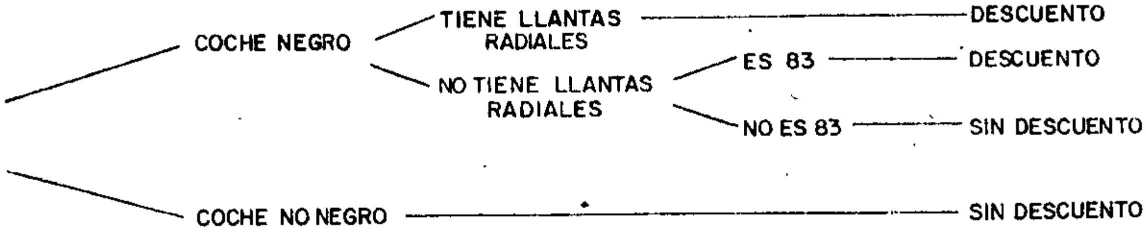
Por medio del DD vamos a poder definir correctamente y sin ambigüedad el significado de un adjetivo.

Arbolès de decisión.- Los árboles de decisión son una representación gráfica para identificar todas las posibles combinaciones y sus acciones. Es una herramienta alternativa del español estructurado y de las tablas de decisión.

En los dos ejemplos anteriores los árboles generados son: para el primero



PARA EL SEGUNDO



En los árboles los nodos intermedios son las condiciones y las hójas (el final) son las acciones.

Tablas de decisión.- Las tablas de decisión son una herramienta que traduce las acciones y condiciones (descritas en el español estructurado o en el narrativò) a una forma tabular. La tabla es difícil de interpretar pero puede ser utilizada como entrada por un algoritmo "manejador de tablas". Estas herramientas permiten introducir orden, regularidad y legibilidad a la especificación funcional.

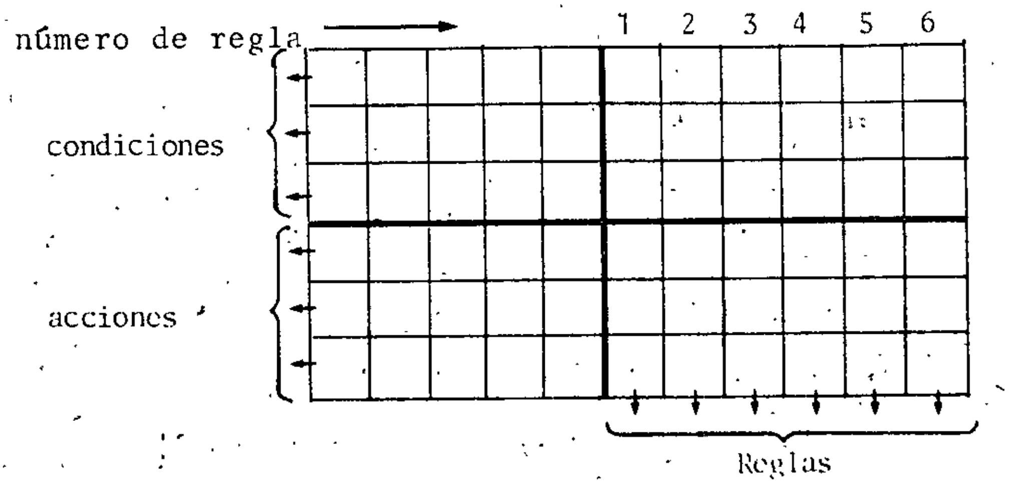
Las tablas se dividen en condiciones, acciones, reglas y número de regla. De esta manera, las diferentes acciones que van a ser tomadas como resultado de las decisiones son listadas en la sección inferior del lado izquierdo, dicha sección recibe el nombre de "stub" de acciones. De esta manera, las diferentes condiciones que afectan las decisiones son listadas en la sección superior del lado izquierdo, llamada "stub" de condiciones. El conjunto de todas las posibles condiciones en la sección superior derecha con sus acciones resultantes ("sí" o "no") en la parte inferior derecha componen las reglas. Por ejemplo la regla 2 es:

		2					
c1:coche negro		Y					
c2:llantas radiales		Y					
c3:modelo 83		N					
a1:descuento		X					
a2:sin descuento							

lo cual es equivalente a decir:

Si el coche es negro y tiene llantas radiales
y no es modelo 83 entonces
tiene descuento (se representa con la X)

De esta manera la tabla queda constituída de la siguiente manera:



Para la construcción de la tabla:

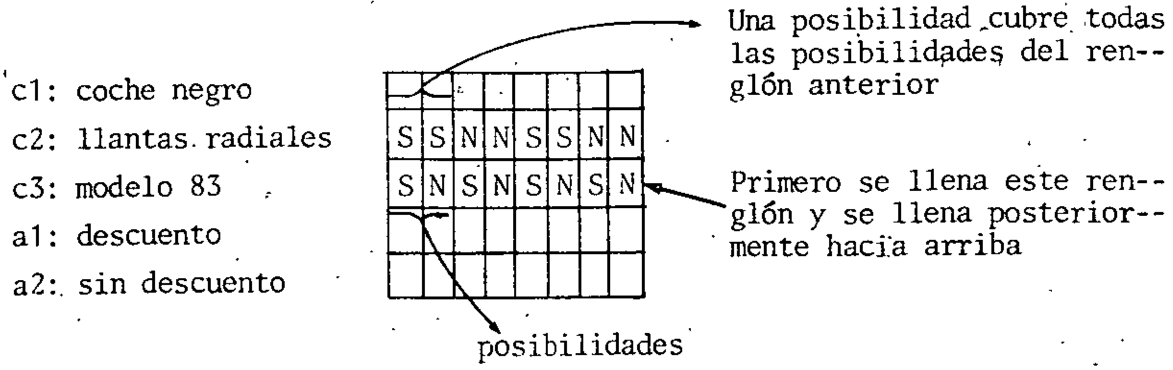
a) Renglones. Se identifican las condiciones y acciones, la suma de ambas proporcionará el número de renglones.

b) Columnas. Se identifican las posibilidades de cada condición y se multiplican. Por ejemplo:

c1:coche negro	2	posibilidades
c2:llantas radiales	x2	"
c3:modelo 8,3	x2	"
total	8	columnas

Note que las columnas son las reglas.

Sección superior derecha. Tome la última condición y alterne todas las posibilidades hasta que termine el renglón. Para el renglón anterior alterne las posibilidades de la condición cada vez que se hayan completado las posibilidades del renglón de referencia. Por ejemplo:


c1: coche negro
c2: llantas radiales
c3: modelo 83
a1: descuento
a2: sin descuento
S S N N S S N N
S N S N S N S N
Una posibilidad cubre todas las posibilidades del renglón anterior
Primero se llena este renglón y se llena posteriormente hacia arriba
posibilidades

Sección inferior derecha. Después de haber cubierto todas las posibles combinaciones se procede a llenar la sección inferior en función de las reglas que se tienen. En algunas ocasiones cuando no se tienen las reglas perfectamente definidas se tiene la oportunidad de consultar nuevamente con el usuario para que la tabla sea llenada adecuadamente. De acuerdo al último ejemplo, la tabla quedaría de la siguiente manera:

	1	2	3	4	5	6	7	8
c1: coche negro	S	S	S	S	N	N	N	N
c2: llantas radiales	S	S	N	N	S	S	N	N
c3: modelo 83	S	N	S	N	S	N	S	N
a1: descuento	X	X	X					
a2: sin descuento				X	X	X	X	X

Una vez que se ha llenado la parte inferior se procede a reducir la tabla de acuerdo a la siguiente técnica:

1.- Encontrar un par de reglas en las cuales:

   + la acción sea la misma,

   + los valores de las condiciones sean los mismos excepto para una y solo una condición que sea diferente.

2.- Reemplace el par de reglas por una sola utilizando el símbolo guión ("-") para la condición que sea diferente.

3.- Repita para el resto de pares de reglas que cumplan con estos criterios.

Después de la reducción la tabla quedaría:

	1/2	3	4/5	6	7/8
c1: coche negro	S	S	-	N	N
c2: llantas radiales	S	N	N	S	N
c3: modelo 83	-	S	N	N	-
a1: descuento	X	X			
a2: sin descuento			X	X	X

Como se observa, los pares de reglas 1-2, 4-5 y 7-8 fueron reducidas ya que, por ejemplo en el par 1-2 sin importar qué modelo sea el coche de todas maneras sí se le hará descuento puesto que las dos primeras condiciones sí se cumplen.

Recomendaciones.-

1. Utilice árboles de decisión cuando el número de acciones sea pequeño y no todas las combinaciones de las condiciones sean posibles; utilice una tabla de decisión cuando el número de acciones sea grande y muchas combinaciones de las condiciones ocurran.

2. Utilice tablas de decisión si duda que sus árboles de decisión muestran totalmente la complejidad del problema.

Comparación de las herramientas.-

Para finalizar, a continuación se muestra una tabla comparativa de las herramientas utilizadas en el proceso de especificación.

Concepto	Español Estructurado	Arbol de Decisión	Tabla de Decisión
Verificación lógica	buena	moderada	muy buena
Legibilidad	buena (todo)	muy buena (sólo para decisión)	moderada (sólo para decisión)
Facilidad de uso	moderada	muy buena	pobre
Facilidad de verificación	pobre	buena	pobre
Especificación del programa	muy buena	moderada	muy buena
Facilidad para la máquina	muy buena	pobre	muy buena
Facilidad de edición	moderada	pobre	muy buena

El español estructurado se utiliza cuando el problema involucra secuencias combinadas de acciones con decisiones o iteraciones.

Los árboles de decisión se utilizan cuando se tiene un número moderado de decisiones que dan como resultado alrededor de 10 a 15 acciones, también se utilizan para representar una tabla de decisión con el objeto de ser más entendible para el usuario.

Las tablas de decisión se utilizan para un gran número de acciones, pero con un número grande de combinaciones de acciones se hace más difícil su manejo.

3.- Especificación de Requerimientos

La especificación de requerimientos es el documento obtenido a partir del análisis. Este documento es generado por el analista y presentado al usuario para su revisión. La especificación de requerimientos servirá como base para el diseño de sistemas, ya que contiene que va a realizar dicho sistema. La especificación contiene básicamente una completa descripción de la información, una detallada descripción funcional, sus criterios de validación y otro tipo de información relacionada.

La organización del documento puede variar de instalación a instalación, pero una manera sencilla y útil de su organización es la siguiente:

1) Introducción
2) Descripción de la Información
    2.1) Diagramas de flujo de datos (DFD)
    2.2) Representación de las estructuras de datos
    2.3) Diccionario de datos (DD)
    2.4) Interfases internas
3) Descripción Funcional
    3.1) Funciones
    3.2) Especificaciones funcionales
    3.3) Restricciones del diseño
4) Criterios de Validación
    4.1) Límites de realización
    4.2) Clases de pruebas
    4.3) Tiempos de respuesta

4.4) Consideraciones especiales

5) Medio Ambiente Operativo

5.1) Equipo

5.2) Software de apoyo

5.3) Interfases del sistema

5.4) Seguridad

5.5) Controles

6) Bibliografía

7) Anexos

La introducción presenta básicamente las metas y objetivos del sistema.

La descripción de la información proporciona una descripción detallada del problema que el sistema va a resolver. Tanto el flujo como la estructura de la información son documentados a detalle.

La descripción funcional presenta los detalles de los procedimientos requeridos para cada función que solucionará el problema. Para la descripción de cada función se utiliza el español estructurado, árboles de decisión o tablas de decisión o una combinación de las tres. Las restricciones del diseño son establecidas y justificadas.

Los criterios de validación actúan como una implicita revisión de la información y de los requerimientos funcionales. En esta sección se deben de especificar qué características debe cumplir el sistema y cómo va a ser probado.

La sección del medio ambiente operativo identifica el equipo requerido y su justificación, así como el software de apoyo y sus interfases para el exitoso cumplimiento del proyecto. Por otro lado, describe la seguridad y privacía impuestos para el sistema, además de sus controles operacionales y las fuentes de dichos controles.

La sección de bibliografía contiene las referencias de todos los documentos relativos al sistema, incluyéndose por ejemplo referencias técnicas.

El anexo contiene información que complementa la especificación.

El documento de especificación de requerimientos debe cumplir una serie de características como las que se mencionan a continuación.

Debe ser altamente mantenible ya que en la mayoría de los casos se requieren modificaciones.

Debe ser altamente entendible, ya que es un vehículo de comunicación entre las partes involucradas.

No debe de tener ambiguedades, ya que esto acarrea problemas para el desarrollo de las siguientes fases.

Debe ser modular, para su fácil acceso y modificación.

4.- Presentación Conceptual

La presentación conceptual es la presentación de la especificación de requerimientos al usuario, donde se realiza una revisión muy minuciosa por parte del analista y del usuario, ya que debe garantizarse que la especificación esté libre de errores, para que de esta manera el resto del proyecto sea exitoso.

La revisión como se indicó debe ser muy minuciosa, contemplándose preguntas como las siguientes:

¿Se han descrito todas las interfases importantes del sistema?

¿Los flujos y estructuras de datos han sido adecuadamente definidos para la solución del problema?

¿Existe claridad en la documentación generada?

¿Existen elementos que describan como se desarrollará el sistema?

¿Los criterios de validación han sido establecidos a detalle?

¿Existen omisiones, inconsistencia o redundancia?

Una vez realizada la revisión y que haya sido aprobado el documento por el usuario, se procede a realizar la siguiente fase: la de arquitectura del sistema.

5.- Herramientas Automáticas

La fase de especificación de requerimientos puede hacer uso de herramientas automáticas, teniéndose las ventajas de ellas. En esta sección mencionarèmos algunas de ellas.

5.1 PSL/PSA.- La herramienta automática PSL/PSA es un paquete de software desarrollado en la Universidad de Michigan por Daniel Teichroew y sus asociados para ayudar en la construcción de los sistemas de información. Dicha herramienta es de gran valor para el analista apoyando en:

+ La descripción de los sistemas de información sin importar el área de aplicación,

+ La creación de una base de datos que contiène los descriptores para el sistema de información,

+ altas, bajas y cambios de los descriptores, y

+ la producción de documentación formateada y diferentes reportes de la especificación.

Los beneficios que proporciona son:

+ Aumenta la calidad de la documentación a través de la estandarización.

+ Coordinación entre los analistas ya que la base de datos es de uso general.

+ Las emisiones e inconsistencias son cubiertas rápidamente a través de referencias cruzadas y reportes.

+ El impacto de las modificaciones es disminuido.

+ Los costos de mantenimiento para las especificaciones son reducidos.

El PSL/PSA consiste en una base de datos que almacena los detalles de la descripción del sistema. Dichos detalles son proporcionados al sistema usando un lenguaje especialmente diseñado para la declaración de problemas. El sistema verifica los datos que llegan en busca de errores e inconsistencias y los almacena en su base de datos. Los procesos y los diccionarios de datos pueden ser producidos automáticamente a través del sistema.

El PSL/PSA, por medio del lenguaje para la descripción del sistema (PSL) permite describir sistemas en un formato que puede ser procesado por la computadora. Una vez que dicha descripción ha sido completada, un analizador de declaración del problema es invocado (PSA). El PSA produce un número de reportes que incluye un registro de todas las modificaciones hechas a la base de datos, el contenido de la clase en diferentes formatos, resúmenes de los reportes para la dirección y análisis de reportes que evalúan las características de la base de datos.

PSL/PSA es más una herramienta de documentación que una herramienta de diseño. Es de un particular valor en los grandes sistemas donde muchos diseñadores y analistas trabajan en las diferentes partes de una aplicación. Con el sistema de datos almacenados, los aspectos del desarrollo pueden ser coordinados y de esta manera evitar inconsistencias.

5.2. Herramientas para apoyar el análisis estructurado.-

El actual aumento en la aplicación de computadoras ha ocasionado una gran demanda en el personal de sistemas, demanda que supera a la oferta. Cuando consideramos esto nos damos cuenta de que es preciso elevar la productividad de todos los empleados y una técnica para lograrlo es proporcionándoles mejores herramientas para realizar sus tareas.

En esta sección se describe un conjunto de herramientas de software que facilitan el trabajo durante la fase de especificación en el ciclo de vida del software. Estas herramientas ayudan a la metodología del análisis estructurado (AE)

El objetivo de estas herramientas es incrementar la productividad (se hace en algunos casos mención de las experiencias habidas en el laboratorio de investigación de Tektronix, Beaverton, Obregon), están basadas en graficación y tienen un significante grado de "inteligencia", ofrecen distintas ventajas al usuario sobre los métodos tradicionales, la más importante es en el área de las comunicaciones, pues el análisis estructurado (AE) proporciona una notación concisa y fácil de usar. Por lo tanto, tan pronto como una idea puede ser identificada, puede ser anotada y comunicada entre los miembros de un grupo, en este sentido un analista puede obtener retroalimentación desde el momento del desarrollo de la idea, en consecuencia con una mejor comunicación y una pronta retroalimentación, el número y la gravedad de los errores, así como el costo en la especificación del software se reducen.

Aun cuando la metodología de análisis estructurado es valiosa, presenta algunos problemas cuando este análisis se realiza a base de papel y lápiz; los cambios a los diagramas son tediosos y el checar la consistencia entre los documentos no es simple y empeora esto conforme el sistema que está modelándose va haciéndose más grande, o cuando se está detallando más, todo esto reduce la productividad de los ingenieros de software y decae la calidad de su trabajo. Sin embargo por medio de algunas herramientas de software pueden solucionarse estas dificultades.

El AE ayuda al analista a controlar la redundancia, eliminar la ambiguedad y asegurar una completa y consistente especificación. No sólo pueden ser detectadas las inconsistencias entre los documentos de AE, es posible también para las herramientas que apoyan este análisis determinar con exactitud los cambios que deben hacerse en otras partes del AE para mantener la consistencia.

Análisis Estructurado Automático.-

El uso del análisis estructurado ha sido analizado para determinar qué funciones pueden ser automáticas.

El propio AE fue utilizado para realizar este análisis. Los beneficios del uso del AE y la experiencia obtenida, proporciona un buen entendimiento de qué herramientas de software son necesarias para apoyar la metodología del AE.

Durante este análisis se determinaron 3 áreas de automatización:

a.- Los graficadores y editores de textos pueden reducir el esfuerzo para modificar un documento de AE.

b.- Varios tipos de errores que un computador puede detectar en un documento de AE fueron identificados.

c.- La consideración de la posibilidad de hacer automático el paso del AE al diseño estructurado.

A continuación se describe el análisis de las herramientas que pudieran ser efectivas para automatizar las áreas mencionadas:

Un editor gráfico de D.F.D. puede ser una herramienta automática efectiva si se dan las consideraciones adecuadas para el soporte gráfico.

La posición automática de los elementos gráficos en un diagrama no es factible porque la posición de estos elementos es crítica para hacer fácil la lectura del D.F.D., y porque no hay métodos heurísticos para determinar cuál debe ser la posición relativa de sus elementos.

En relación al diccionario de datos y las especificaciones estructuradas, pueden ser creados y modificados usando cualquier editor de textos disponible.

El editor de D.F.D. requiere de una terminal graficadora para despliegue interactivo, así como dispositivos de copiado para revisar y actualizar los diagramas.

De la evaluación de las funciones se concluye que unas pueden ser automatizadas y otras no. Una computadora puede checar de manera efectiva errores que pueden describirse con reglas estrictas y breves, pero una persona es más efectiva en detectar errores que requieren entender la semántica de un documento de AE, por ejemplo: una computadora puede detectar usos inconsistentes de un flujo de datos entre varios D.F.D. jerárquicos relacionados. Una persona es mucho mejor para determinar si un flujo de datos está bien nombrado o no.

Puesto que los documentos de análisis estructurado pueden ser representados en una computadora, una herramienta de evaluación puede detectar inconsistencias entre documentos. Sin embargo, la detección de errores no es el camino más satisfactorio para mantener la consistencia.

Cuando un conjunto consistente de documentos de Análisis Estructurado es construido, esta consistencia será alterada sólo cuando uno de los documentos se modifique. Si estos cambios son monitoreados algunos tipos de errores pueden ser evitados por la derivación automática de los cambios a otros documentos del análisis estructurado.

Cuando el analista ha terminado la especificación, el siguiente paso es el diseño del sistema.

El diseño estructurado (D.E.) se realiza siguiendo el análisis estructurado, el conjunto de D.F.D. puede ser traducido en gráficas de la estructura preliminar. Una herramienta automática que realice esta traducción puede ser la interfase con las herramientas de apoyo para diseño estructurado; dado que el paso del A.E. al D.E. requiere de la intervención humana, esta fase no puede ser totalmente automática. El desarrollo de un traductor semiautomático efectivo es aún un desafío.

En resumen, el análisis de la metodología del análisis estructurado desarrollado en el medio ambiente de ingeniería de Tektronix, concluye que las herramientas para aplicar esta metodología deben tener las siguientes capacidades:

1. Modificar y desplegar documentos del A.E.
2. Checar errores en documentos del A.E.
3. Ayudar a mantener la consistencia de los documentos del A.E.

Herramientas de Apoyo para el Análisis Estructurado.-

Las herramientas de apoyo al A.E. están escritas en Modula-2 y corren bajo UNIX V7. Una terminal Tektronix 4014 es usada para gráficas interactivas, estas herramientas pueden ser usadas individualmente o combinadas, el usuario puede adecuarlas a sus necesidades. Algunas consideraciones al respecto se escriben a continuación:

1. Cada herramienta realiza una sola función exáctamente definida.

2. Las interfases de todas las herramientas son sencillas.

3. Las herramientas no asumen nada acerca de la fuente u origen de los datos que pasan a través de ellas.

4. Las herramientas están organizadas en familias (esto es, la función de la herramienta es repetida por cada documento del Análisis Estructurado).

5. Todas las interfases para una familia de herramientas son consistentes.

Además de la capacidad de adecuar estas herramientas de soporte, los usuarios pueden expanderlas añadiendo sus propias herramientas.

Las funciones realizadas por cada familia de herramientas de soporte son descritas a continuación:

. El editor modifica interactivamente un documento de análisis estructurado y deriva los cambios requeridos, eso ayuda a mantener la consistencia de los documentos.

. Las herramientas de evaluación detectan errores en un documento.

. El formateador convierte la representación interna de un documento de A.E. a una forma cómoda de observar.

. Las herramientas de depuración (clean-up) ayudan a mantener la consistencia entre los documentos por validación, haciendo los cambios que han sido realizados por medio del editor.

Hay dos tipos de editores:

. Editor D.F.D.

. Editor A.E.

El Editor de Diagramas de Flujos de Datos es interactivo para gráficas, usado para crear o modificar un D.F.D., un dispositivo graficador es usado para especificar la posición de cada uno de los elementos en el D.F.D.

El Editor A.E. modifica la estructura del análisis estructurado o hace cambios cuyos efectos se sienten a lo largo de todo el análisis. Usando el Editor A.E., un D.F.D. descendente o las especificaciones pueden conectarse o desconectarse de algunas partes del proceso principal. También estos editores se usan para hacer cambios en una porción del A.E., por ejemplo, cuando un nombre de dato es cambiado.

Cada comando que cambia el A.E. o un D.F.D. es analizado por el Editor para generar los cambios que ayudarán al usuario a mantener la consistencia en todo el conjunto de documentos.

En cuanto a las herramientas de evaluación para checar errores, podemos decir que hay 4, una para cada uno de los elementos del A.E., y una para todo el conjunto. Estas herramientas detectan 2 tipos de errores:

a.- "hard errors" como datos con nombres indefinidos o detección de archivos para lectura únicamente, este tipo de errores indican una violación de las reglas de la metodología.

b.- "soft errors": estos errores perjudican las reglas de el metodología como es el máximo número de procesos que son permitidos en un D.F.D. Usualmente estos errores indican que el problema que el analista está tratando de resolver, usando A.E., no está bien entendido.

Los formateadores transforman el D.F.D., las especificaciones o el diccionario de datos en representaciones fácilmente observables. La salida es desplegada en terminales graficadoras, en éstos se asocian elementos gráficos a cada uno de los elementos del diagrama de flujo de datos. Este tipo de herramienta no posiciona automáticamente los elementos gráficos, más bien, el usuario especifica la posición de cada elemento gráfico por medio del Editor.-D.F.D.

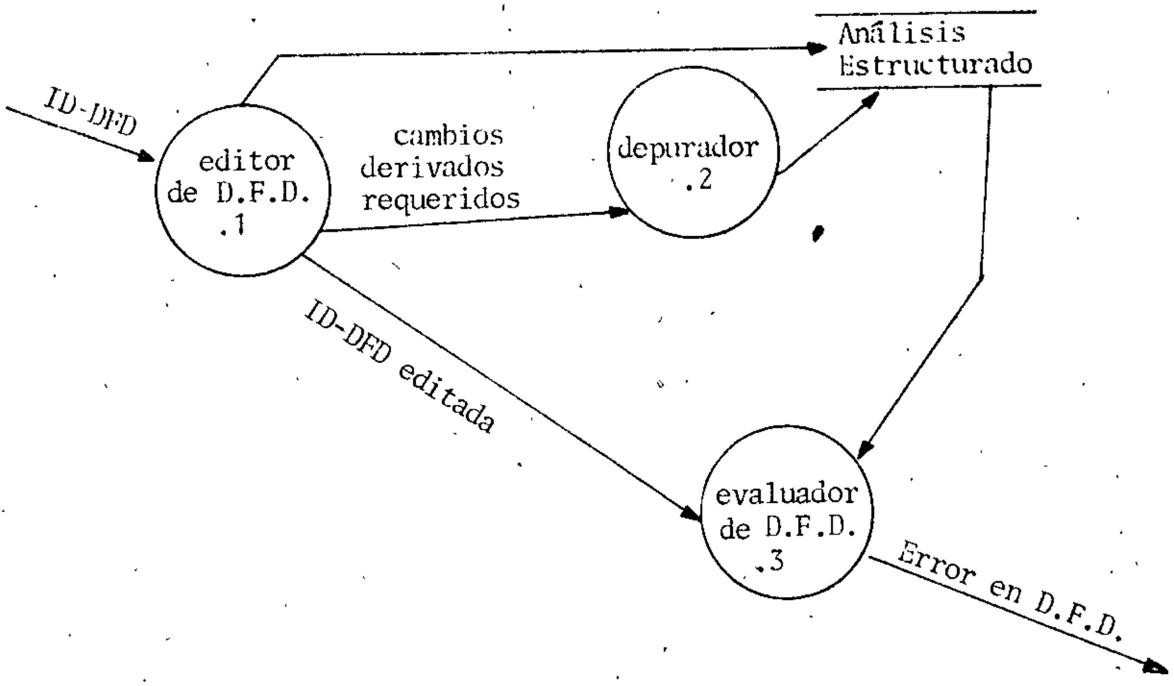
El formateador D.F.D. automáticamente posiciona textos asociados a los elementos gráficos; p.e. un nombre y número de proceso es encerrado en un círculo. El formateador del diccionario de datos también ordena la definición de los datos jerárquica y alfabeticamente. Los nombres usados con una definición pueden ser listados en 2 ó más lugares. Primero la definición de datos aparece en orden alfabético. Copias adicionales son colocadas inmediatamente después de donde el nombre del dato ha sido referenciado.

Para mantener la consistencia respecto a los documentos del A.E. se utiliza un depurador (clean-up), debido a las relaciones entre los documentos, un cambio en uno de éllos puede afectar a otros, por ejemplo, si un proceso es borrado del

D.F.D., el depurador borra cualquier especificación (mini-spec) o D.F.D. que sea dependiente del proceso borrado del diccionario de datos. El editor deriva estos cambios y el depurador verifica la necesidad del cambio.

Las herramientas para soportar el A.E. pueden ser vinculadas en distintas formas. A continuación se describe como éstas interactúan para realizar funciones de edición/depuración/ evaluación.

Las herramientas son diseñadas para trabajar juntas, lo cual provee un mayor poder en sus funciones. Estas combinaciones no sólo modifican un D.F.D., sino que lo checan de errores y ayudan a mantener la consistencia. Cada tipo de herramientas asume responsabilidad sobre cierto tipo de errores, algunos son rechazados por el editor, otros prevenidos por el depurador y algunos más reportados al usuario por herramientas de evaluación.



El usuario edita un D.F.D. de manera interactiva. La sesión de edición procede así: ¿las salidas derivan los cambios requeridos? que son descripciones de cambios hechos a este D.F.D. y que afectan a otras partes del A.E.. Cuando la edición es completada el depurador (clean-up) lee los cambios requeridos y los realiza para mantener la consistencia.

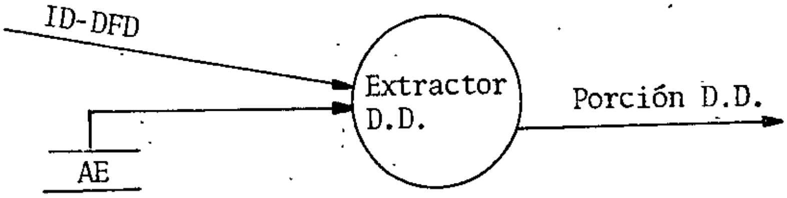
Una vez que el depurador ha terminado, el evaluador de D.F.D. puede empezar, evalúa el diagrama y reporta al usuario los errores encontrados, el usuario deberá decidir la forma de corregirlos, hay otro tipo de errores que sí puede resolver el evaluador por ejemplo: cuando a un prceso se le cambia de número, los elementos que de éste dependen son también cambiados.

El editor de D.F.D. mantiene la validez de ciertas reglas:

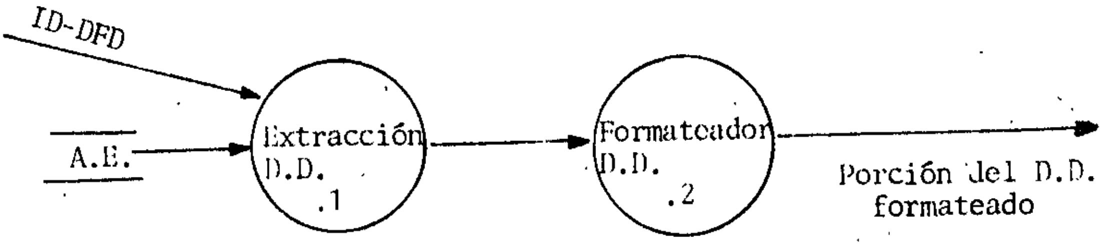
- Cada flujo de datos debe tener un origen y un destino.
- Si el origen y el destino son borrados, también se borra el flujo de datos.
- Las eliminaciones de elementos u otros cambios afectan al D.F.D. que está siendo editado.
- Los cambios que afectan el D.F.D. en edición son hechos inmediatamente, no pasan a la fase de depuración.

Ampliando las herramientas de A.E.-

Si las herramientas de A.E. estuvieran siendo usadas en un gran proyecto y un D.F.D. debiera ser revisado, se ahorraría un gran tiempo si para esta revisión no tuviera que buscarse en todas las páginas del diccionario de datos del sistema. Sería preferible un diccionario de datos listando sólo los términos necesarios para la revisión particular del D.F.D., una herramienta con estas características pudiera denominarse "extractor del Diccionario de Datos".



Esta herramienta de extracción recupera el D.F.D. de interés, da una lista de términos que son usados, los extrae del diccionario de datos (D.D.) y las coloca en una porción de diccionario de datos. Este extractor puede ser combinado con otras herramientas que apoyan el A.E. como a continuación se muestra:



La simple interfase del formateador de D.D. hace posible determinar cuál debe ser la salida del extractor. Los tipos de datos abstractor para el D.F.D. y para el Diccionario de Datos hacen una tarea fácil el extraer la información deseada. La adaptabilidad de las herramientas de apoyo al A.E. ha sido una importante consideración en su diseño e implementación.

Medición de la Efectividad de las H.A.'s para apoyar el Análisis Estructurado.-

El propósito de las herramientas mencionadas es incrementar la productividad en la ingeniería de software, pero medir qué tan bien han cumplido su propósito es un paso difícil, pues las herramientas de apoyo al A.E. son nuevas en Tektronix y no hay datos suficientes para argumentar convincente-- mente su efectividad aunque una observación informal sugiere un incremento en la productividad. Antes de que hubiera disponibilidad de estas herramientas uno de los mayores cuellos de botella en un proyecto era el dibujo de los D.F.D.

Una consideración más que debe tenerse presente con objeto de la evaluación es que cuando los diagramas son entregados para su revisión hay una tendencia a pensar que como se ven bien presentados, éstos deben estar correctos necesariamente y no se les presta suficiente atención a los detalles. Esta tendencia se debe a que se sabe que las herramientas hacen algunas evaluaciones.

El análisis estructurado ha sido usado exitosamente por va rios años, el uso de herramientas automáticas quita a los analistas el realizar tareas tediosas que propiciaban errores frecuentes. Con estas herramientas pueden crearse, modificarse y evaluarse documentos relacionados con el análisis estructurado ayudando a mantener la consistencia de estos últimos.

Las funciones básicas para soporte en A.E. han sido realizadas en un conjunto de herramientas simples. Los usuarios pueden ampliar la capacidad de éstas usándolas combinadamente con otras funciones de utilería (utility) ya existentes, o por medio de la construcción de nuevas.

El uso de estas herramientas producirá con seguridad un significante incremento en la productividad y en una amplia variedad de caminos en la ingeniería y en el desarrollo de los ambientes de sistemas de información.

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

DOCUMENTACION DE PROYECTOS EN INFORMATICA

DIAGRAMAS WARNIER-ORR

(DOCUMENTACION BASICA)

FEBRERO, 1984

vi. DIAGRAMAS WARNIER-ORR

(DOCUMENTACION BASICA)

FUNDAMENTO DE DISEÑO WARNIER/ORR

El diseño lógico mediante diagramas Warnier/orr, está basado en tres observaciones producto de diseño correcto, estas son:

DISEÑO LOGICO WARNIER/ORR {

- Diseño orientado a salidas
- Diseño Lógico antes que físico
- Diseño basado en estructuras de datos

Diseño orientado a las salidas

La frase "orientado a salidas" es sinónimo de "orientado a resultados", esto sugiere que un buen diseño debe comenzar con un claro y completo entendimiento de las salidas, antes de intentar producir módulos o programas.

Diseño lógico antes que físico

La segunda observación se relaciona directamente con las preguntas, qué se quiere hacer ?, antes de, como hacerlo?, es decir, la primera pregunta profundiza en los aspectos lógicos, y la segunda en las características físicas.

Por LOGICO entenderemos todo lo independiente del Hardware. Y por FISICO los aspectos específicos del ambiente Hardware.

Las ventajas de razonar de este modo tienen un beneficio inmediato al evitar gastos innecesarios, incluyendo el uso del computador, puesto que el diseño se concentra en soluciones lógicas que muchas veces muestran viabilidades importantes; como el seguir con un procedimiento manual a base de calculadoras, máquinas de escribir y personal de oficina, o simplemente existen partes en el sistema cuya automatización tendría un costo elevado y es muy posible que con una organización adecuada se solucione.

Por otra parte, el ambiente físico donde reside el Software es altamente cambiable; por ejemplo: cambios de computador, cambios de métodos de acceso, reorganización de archivos, - cambios en los lenguajes de programación, o en el sistema - operativo.

Pero si realmente el problema se encuentra diseñado logicamente, ante cualquier cambio la respuesta sería fácil.

Además, cuando el diseñador tiene en mente restricciones físicas, esto origina complicaciones que alargan la solución al problema, y no le permiten meditar en otros satisfactores.

Diseñar es una fase obligatoria antes de codificar y probar, es independiente de cualquier máquina.

DISEÑO BASADO EN ESTRUCTURAS DE DATOS

Este principio fué descubierto en forma independiente por Warnier (Francia), y Michel Jackson (Inglaterra). La investigación de ambos se centró en la representación de la estructura de buenos programas, concluyendo que la organización de un buen programa es reflejo de la organización de datos que serán procesados.

Un axioma derivado de lo anterior en el metodo warnier/orr dice:

> Si la organización de los datos puede ser descubierta y representada, entonces la organización del programa puede ser totalmente derivada.

DIAGRAMAS WARNIER/ORR

Descripción del Instrumento:

Un diagrama es una forma de representar la lógica de la solución antes de proceder a escribir programas, quizá de los medios mas conocidos para hacer esto, el diagrama de flujo sea el mas popular, por supuesto existen otros medios como; las cartas estructuradas, cartas Nassi/Shneiderman/Chapin, pseudocodico, tablas de decisión y diagramas Hipo.

Pudieramos decir que el diagrama warnier/orr, es una clase de "super diagrama de flujo", que permite mostrar:

ESTRUCTURAS DE FUNCIONES/FLUJO DE DATOS
ESTRUCTURAS DE DATOS
ESTRUCTURAS DE PROGRAMAS

Este instrumento, es básicamente lógico y no está relacionado con ningún proveedor de equipo de computo, o con algún lenguaje de programación.

Requiere básicamente:

- EXPRESIONES EN LENGUAJE NATURAL (ESPAÑOL, FRANCES, INGLES...)
- PAPEL
- LAPIZ

Y construcciones básicas sencillas acerca de los datos y programas.

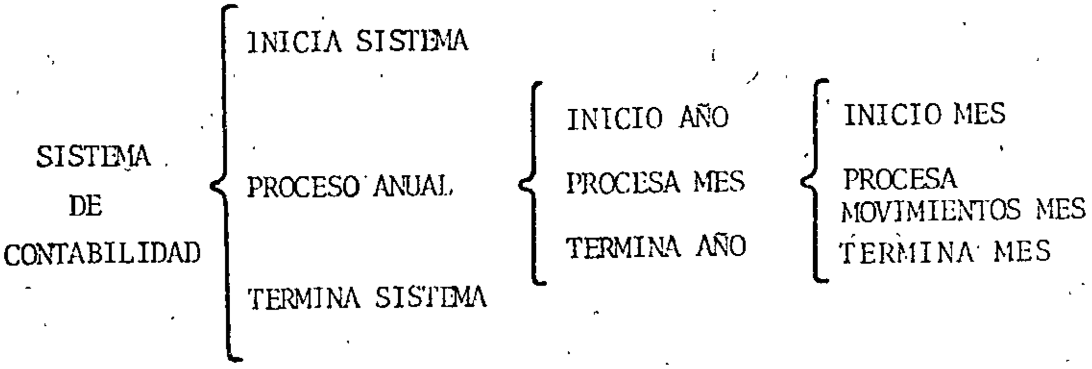
Esto presenta una economía importante cuando se plantean soluciones a problemas de información, y estos requieren ser documentados y la calidad debe ser alta.

NOTACION

La notación básica de este instrumento son las LLAVES, las cuales sirven para formar conjuntos; de datos, información, acciones o procesos.

Los diagramas warnier/orr tienen su fundamento en las matematicas, precisamente en la parte de teoría de conjunto.

Del lado izquierdo de la llave y precisamente en el centro se comienza por describir el conjunto principal, mientras que a la derecha se enumeran los elementos mismos que pueden ser conjuntos nuevamente, tal como se comprende en teoría de conjuntos, los conceptos de subconjuntos.

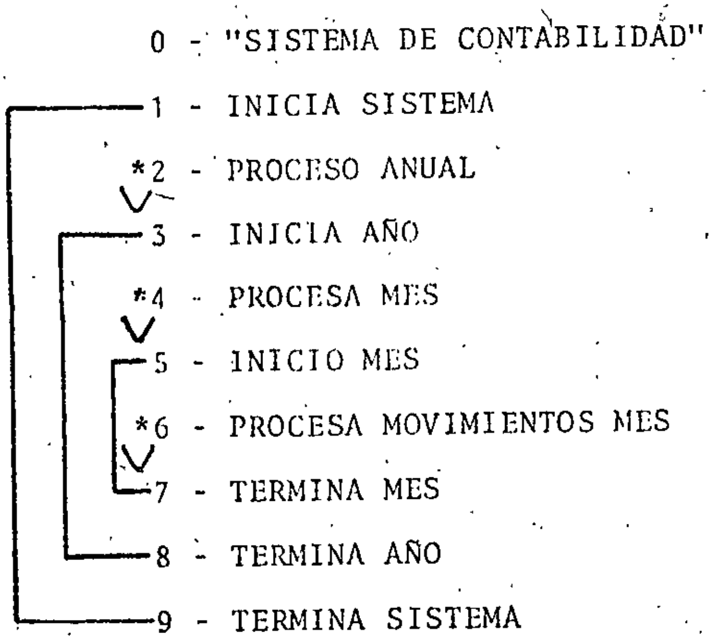


En el caso del conjunto "Sistema de Contabilidad" (equivalencia de la llave), este comienza diciendo CONTIENE A los conjuntos "Inicia Sistema", "Proceso anual", "Termina Sistema"; el conjunto "proceso anual, CONTIENE A, "Inicia Año", "Procesa Mes", "Termina Año" y asi sucesivamente.

Invariablemente las llaves pueden significar CONTIENE A o CONSISTE DE.

Notaremos también que la lectura del diagrama, comienza a ser leído de arriba hacia abajo, leyendo los conjuntos intermedios antes de proseguir a otro conjuntó.

En el conjunto mostrado, nuestro proceso de lectura consiste de los siguientes pasos.



0 - "SISTEMA DE CONTABILIDAD"

1 - INICIA SISTEMA

*2 - PROCESO ANUAL

3 - INICIA AÑO

*4 - PROCESA MES

5 - INICIO MES

*6 - PROCESA MOVIMIENTOS MES

7 - TERMINA MES

8 - TERMINA AÑO

9 - TERMINA SISTEMA

Las líneas de ejemplo anterior muestran la correspondencia entre conjuntos.

De hecho poco a poco resulta familiar y altamente comprensivo el diagrama warnier/orr.

Otras notaciones importantes que nos ayudarán posteriormente a utilizar el instrumento son:

$$\text{A Ñ O} \left\{ \begin{array}{l} \text{MES} \\ \ (12) \end{array} \right.$$

El conjunto "AÑO" <u>CONTIENE A</u> mes (12) veces, esto sucede cuando se conocen cuantos elementos forman el conjunto.

$$\text{ARCHIVO} \left\{ \begin{array}{c} \text{REGISTRO} \\ (0,R) \end{array} \right.$$

Significa que el conjunto "ARCHIVO" <u>CONTIENE A,</u> cero o (R) registros.

$$\text{EMPRESA} \left\{ \begin{array}{c} \text{EMPLEADO} \\ (1,E) \end{array} \right.$$

Significa que el conjunto "EMPRESA" <u>CONTIENE A,</u> uno o (E) empleados.

O mas brevemente:

$$\text{EMPRESA} \left\{ \begin{array}{c} \text{EMPLEADO} \\ (E) \end{array} \right.$$

Cuando un conjunto contiene a, precisamente un conjunto, este se puede representar por:

$$\begin{array}{c}\text{SISTEMA}\\ \text{X}\end{array}\left\{\begin{array}{c}\text{INICIA SISTEMA}\\ (1)\\[1em] \text{PROCESA SISTEMA}\\ (1)\\[1em] \text{TERMINA SISTEMA}\\ (1)\end{array}\right.$$

O simplemente como:

$$\begin{array}{c}\text{SISTEMA}\\ \text{X}\end{array}\left\{\begin{array}{l}\text{INICIA SISTEMA}\\[1em] \text{PROCESA SISTEMA}\\[1em] \text{TERMINA SISTEMA}\end{array}\right.$$

Un simbolo de gran utilidad es, ⊕ que se utiliza para representar exclusiones de datos o bien, de acciones.

$$\text{SEXO EMPLEADO}\left\{\begin{array}{c}\text{FEMENINO}\\ \oplus\\ \text{MASCULINO}\end{array}\right.$$

En este ejemplo, el conjunto "SEXO EMPLEADO" puede ser o femenino o masculino exclusivamente, nunca los dos.

$$\text{MOVIMIENTO CONTABLE} \left\{ \begin{array}{c} \text{CARGO} \\ \oplus \\ \text{ABONO} \end{array} \right.$$

El conjunto "MOVIMIENTO CONTABLE", puede ser o cargo o abono, nunca ambos.

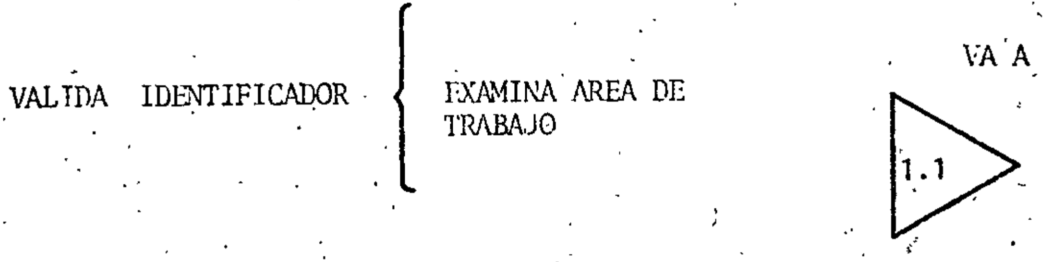
Un simbolo lógico, que sirve para excluir y aceptar al menos un conjunto, se representa por:

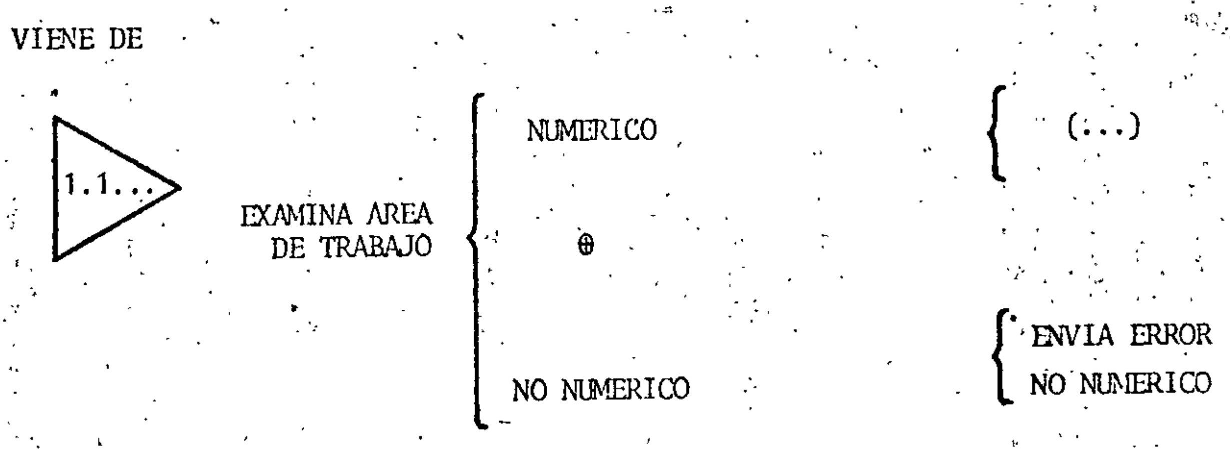
$$\begin{array}{c}\text{PROCESA} \\ \text{TRANSACCION}\end{array} \left\{ \begin{array}{c} \text{ALTA} \\ (0,1) \\ \oplus \\ \text{BAJA} \\ (0,1) \\ \oplus \\ \text{CAMBIO} \\ (0,1) \\ \oplus \\ \text{CONSULTA} \\ (0,1) \end{array} \right.$$

En el ejemplo anterior, representamos que el "PROCESA TRANSACCION", puede ocurrir una alta o no, una baja o no, un cambio o no, una consulta o no.

El simbolo (o,1), significa que ocurre cero a una vez.

Cuando se desea hacer una referencia a un AUMENTO de la misma página o fuera de ella, se puede hacer uso de una pequeña llave con alguna numeración que represente el número de página y la sección dentro de la página.





Observemos también que en el caso donde se ignore el conjunto podemos hacerlo mediante el uso de (....) que equivale a conjunto vacío, o a la expresión CONTINUA A otro conjunto, o bien se puede usar la expresión NULO, que significa lo mismo.

En los diagramas warnier/orr para representar, procedimientos, acciones, procesos esto se debe hacer formando expresiones iniciadas, mediante un verbo imperátivo, como por ejemplo:

ACEPTA
ABRE
ACOMODA
ACTUALIZA
ANALIZA
ARCHIVA
BORRA
BUSCA
CALCULA
CAPTURA
CIERRA
CLASIFICA
CODIFICA
COLOCA
COMPRUEBA
CONSULTA
CONTROLA
COPIA
CRITICA
DESPLIEGA
DISTRIBUYE
EDITA
EJECUTA
ESCRIBE
EXAMINA
FORMATEA
GENERA
GRABA
INCREMENTA
INSERTA
LEE
ETC.

Seguido de un complemento directo.

Ejemplo:

GENERA BASE DE DATOS
CAPTURA CUESTIONARIO
CALCULA DESCUENTO
VALIDA TRANSACCION
RECHAZA FACTURA
ETC.

En el caso de representar datos esto puede hacerce formando sustantivos seguidos de un complemento.

CONSTRUCCIONES BASICAS WARNIER/ORR

Los pequeños ladrillos permiten realizar grandes construcciones; de manera analoga, es posible pensar que la informática debe estar basada en algunas construcciones que faciliten expresar problemas informáticos, es, decir relaciones que asocien datos y procedimientos, con estructuras tan simples y sencillas como son los ladrillos.

En warnier/orr se definen cuatro construcciones básicas, mediante las cuales es posible representar la mayoría de las soluciones informáticas de manera organizada y sistemática.

CONSTRUCCIONES BASICAS WARNIER/ORR {
- JERARQUIAS
- SECUENCIAS
- REPETICIONES
- SELECCIONES

Jerarquías

Es el tipo de construcción que permite asociar conjuntos complejos a partir de un conjunto simple, mediante un proceso lógico de dividir sistematicamente hasta obtener un resultado lo bastante descriptivo como para representar instrucciones en algún lenguaje de programación, o bien un conjunto de funciones lógicas donde se muestre una macro solución, o una agrupación de datos que sea, un reporte o un conjunto de archivos organizados (Base de datos).

El objetivo de la jerarquización es dividir grandes problemas en pequeños subproblemas.

En los diagramas warnier/orr, se deben identificar los niveles que componen la jerarquía y proceder sistematicamente hasta -- llegar a una solución satisfactoria.

$$\begin{array}{c}\text{REGISTRO DEL}\\ \text{EMPLEADO}\end{array}\left\{\begin{array}{l}\text{IDENTIFICADOR EMPLEADO}\\ \text{NOMBRE EMPLEADO}\\ \text{DIRECCION EMPLEADO}\\ \text{SALARIO EMPLEADO}\end{array}\right.$$

El conjunto de datos "REGISTRO DEL EMPLEADO" CONSISTE DE, un identificador empleado, nombre empleado, dirección empleado, salario empleado, esto también significa una jerarquía de un solo nivel, sistematicamente para describir mas niveles tendríamos que descomponer hasta donde sea comprendido claramente el problema, del ejemplo anterior derivamos una mejor observación como:

- REGISTRO DE EMPLEADO
  - IDENTIFICACION EMPLEADO
  - NOMBRE EMPLEADO
    - Apellido paterno
    - Apellido materno
    - Nombres
  - DIRECCION EMPLEADO
    - Calle
    - Ciudad
    - Estado
    - Código postal
  - SALARIO EMPLEADO

En este caso el conjunto "Registro de Empleado", tiene dos niveles jerárquicos, un nivel contiene a Identificador Empleado, nombre empleado, dirección empleado y salario empleado, el segundo nivel referido al conjunto "Nombre empleado", contiene a, "Apellido paterno", "Apellido materno" y "Nombres".

Un ejemplo descriptivo de como se efectúa la integración del costo de la producción sería:

- COSTO PRODUCCION
  - COSTO PRIMO
    - MATERIALES DIRECTOS
      - MATERIALES IDENTIFICADOS EN EL PRODUCTO
    - MANO OBRA DIRECTA
      - TRABAJADORES RELACIONADOS A MANEJAR LOS MATERIALES DEL PRODUCTO
  - COSTO INDIRECTO
    - MATERIALES INDIRECTOS
      - ENERGIA MAQUINAS
      - LUBRICANTES
      - HERRAMIENTAS
      - EMPAQUES
    - MANO DE OBRA INDIRECTA
      - GERENTE DE PRODUCCION
      - SUPERVISORES
      - TECNICOS
    - GASTOS INDIRECTOS DE PRODUCCION
      - DEPRECIACIONES
      - ENERGIA
      - RENTA
      - IMPUESTOS
      - SEGUROS

Dando como resultado un diagrama warnier/orr de tres niveles.

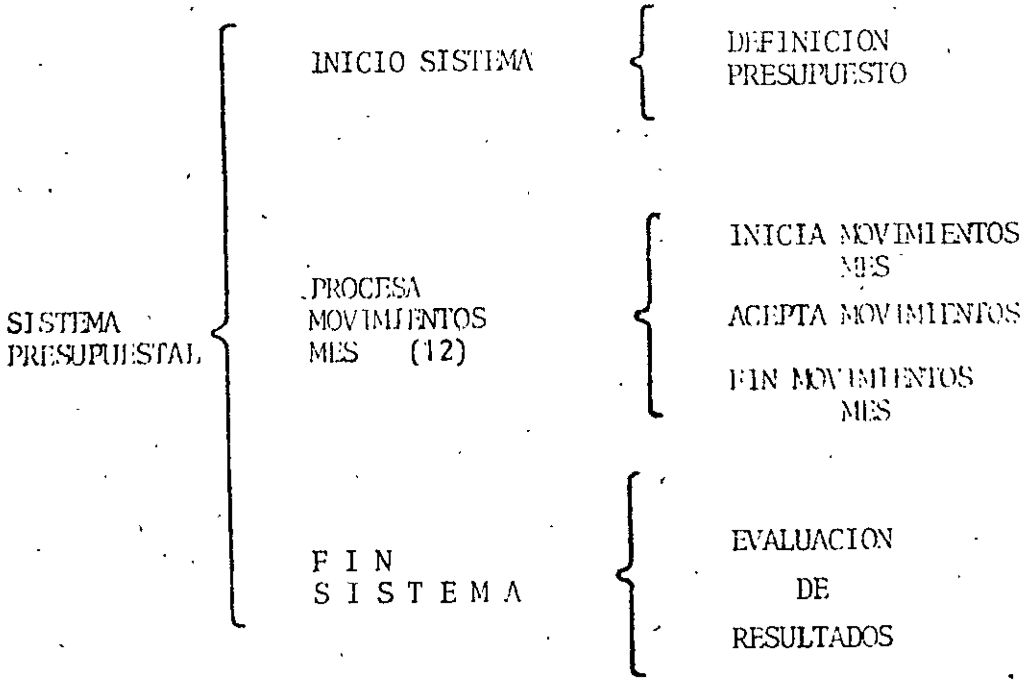
Por mucho tiempo la utilización de llaves a sido utilizada, por Pedagogos, Biólogos, Historiadores, etc. para clarificar la descomposición o ilustración de conceptos. De hecho la informática está relacionada directamente con conceptos de información y es, después de los seres humanos lo mas importante de una organización.

Secuencias

La segunda construcción básica necesaria para diseñar es la secuencia, siendo esta estructura una habilidad para colocar conceptos, cosas, objetos, sujetos, de tal manera que ocurran unos después de otros.

La secuencia es la más simple de las cuatro construcciones y es la forma mas comunmente usada de cualquier problema, después de partir o analizar datos o procedimientos, sigue la secuenciación. En el caso del conjunto "Registro Empleados" la secuencia se muestra mediante la posición de los elementos, iniciando con identificador empleado, SEGUIDO POR, nombre empleado, SEGUIDO POR, dirección empleado, SEGUIDO POR, salario empleado, aquí termina.

Un ejemplo de secuenciación, puede darse con respecto a un -- sistema presupuestal:



En el presente diagrama la secuencia del conjunto "Sistema Presupuestal", consiste de "Inicio sistema", SEGUIDO DE "Procesa movimientos mes", SEGUIDO DE, "Fin sistema".

Repeticiones

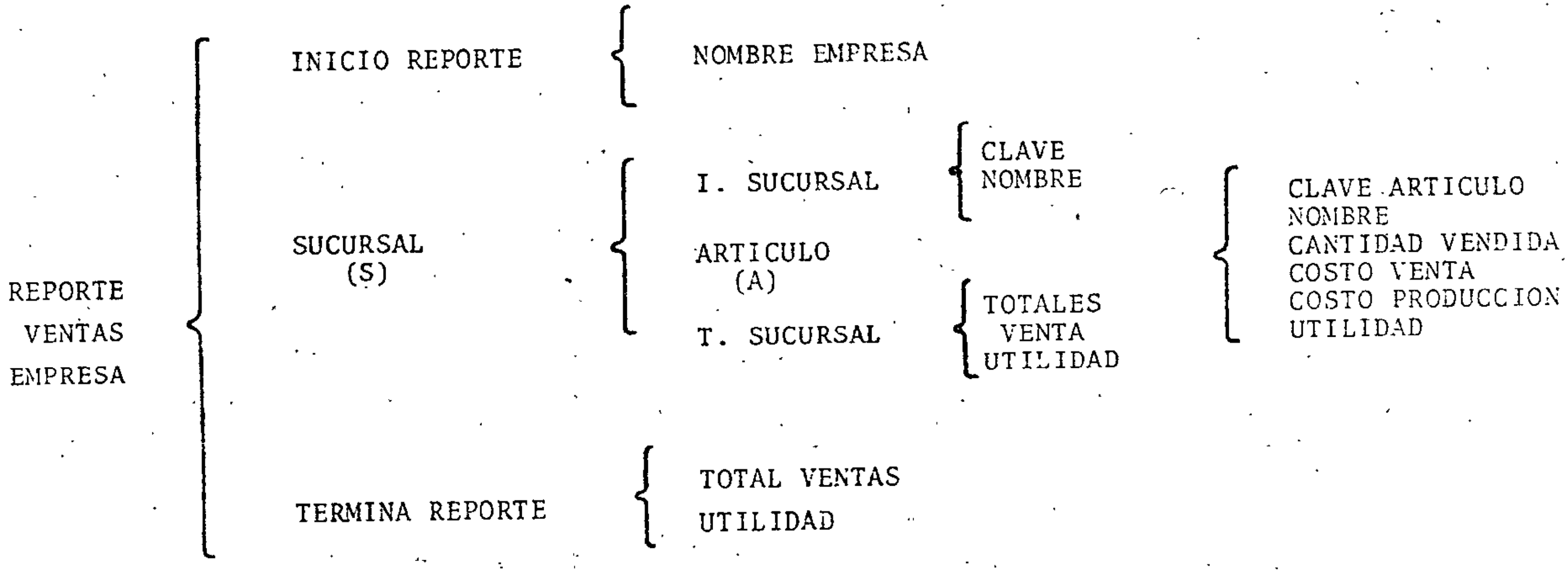
La tercera construcción se denomina repetición, esta consiste de representar datos o acciones que ocurren repetidas veces, se define uno y continúa otra vez, esto es una simplificación abstracta de una secuencia. Cuando se trata de acciones esto es equivalente a los ciclos, cuando se trata de datos, estos pueden ser registros de un archivo, arreglos de datos vectoriales o matricales etc.

$$\begin{array}{c}\text{ARCHIVO}\\ \text{EMPLEADOS}\end{array} \left\{ \begin{array}{lcl} \text{REGISTRO} & 1 & \text{EMPLEADO} \\ \text{REGISTRO} & 2 & \text{EMPLEADO} \\ \text{REGISTRO} & 3 & \text{EMPLEADO} \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ \text{REGISTRO} & \text{E} & \text{EMPLEADO} \end{array} \right.$$

La anterior secuencia puede representarce con una construcción repetitiva como:

$$\text{ARCHIVO EMPLEADO} \left\{ \begin{array}{c} \text{REGISTRO EMPLEADO} \\ (\text{E}) \end{array} \right.$$

El símbolo (E) indica que existe, registro empleado; E veces, cuando esto se hace, se debe a que desconocemos la cantidad de registros que puede tener el archivo (1, 300, 900, 1500 - - - -).



REPORTE VENTAS EMPRESA

INICIO REPORTE

NOMBRE EMPRESA

SUCURSAL (S)

I. SUCURSAL

CLAVE
NOMBRE

ARTICULO (A)

CLAVE ARTICULO
NOMBRE
CANTIDAD VENDIDA
COSTO VENTA
COSTO PRODUCCION
UTILIDAD

T. SUCURSAL

TOTALES
VENTA
UTILIDAD

TERMINA REPORTE

TOTAL VENTAS
UTILIDAD

En el diagrama warnier/orr, deseamos representar el proceso de contabilizar los artículos vendidos por una empresa, organizada en sucursales, para analizar los artículos que más demanda tienen.

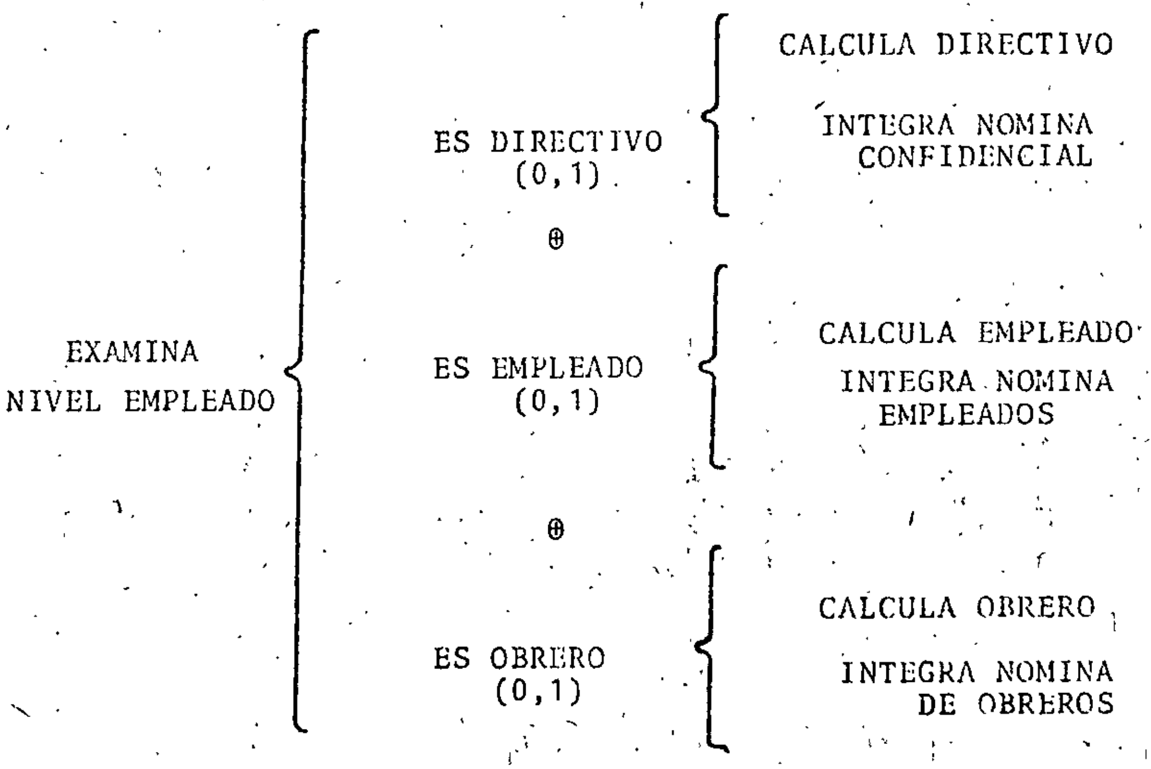
En este ejemplo, el conjunto sucursal sucede (S) veces y cada sucursal CONTIENE A, un inicio de sucursal y artículos, los cuales suceden (A) veces, finalmente existe una terminación de sucursal.

Alternación

También se denomina a esta construcción selección, su aplicación consiste en particionar un conjunto en dos o mas alternativas, según el caso, equivalente a la estructura case de programación estructurada, en warnier/orr esta estructura es mas universal porque incluye la representación de datos y procedimientos.

En este tipo de construcciones se hace uso del símbolo ⊙, el cual equivale a un conectivo lógico o exclusivo, es decir, que solo uno de los casos sucede, nunca dos a la vez.

En un análisis se observa que una nómina tiene que obtenerse en tres formas, nómina confidencial, donde entran los directivos, nómina empleados, para los técnicos y personal administrativo y la nómina de obreros, básicamente compuesta por el personal que participa directamente en la producción; de este hecho al exáminar a los empleados, básados en el nivel de -- sueldo o categoría, decidimos la clase cálculo que le toca.



EXAMINA NIVEL EMPLEADO

ES DIRECTIVO (0,1)

CALCULA DIRECTIVO

INTEGRA NOMINA CONFIDENCIAL

⊕

ES EMPLEADO (0,1)

CALCULA EMPLEADO

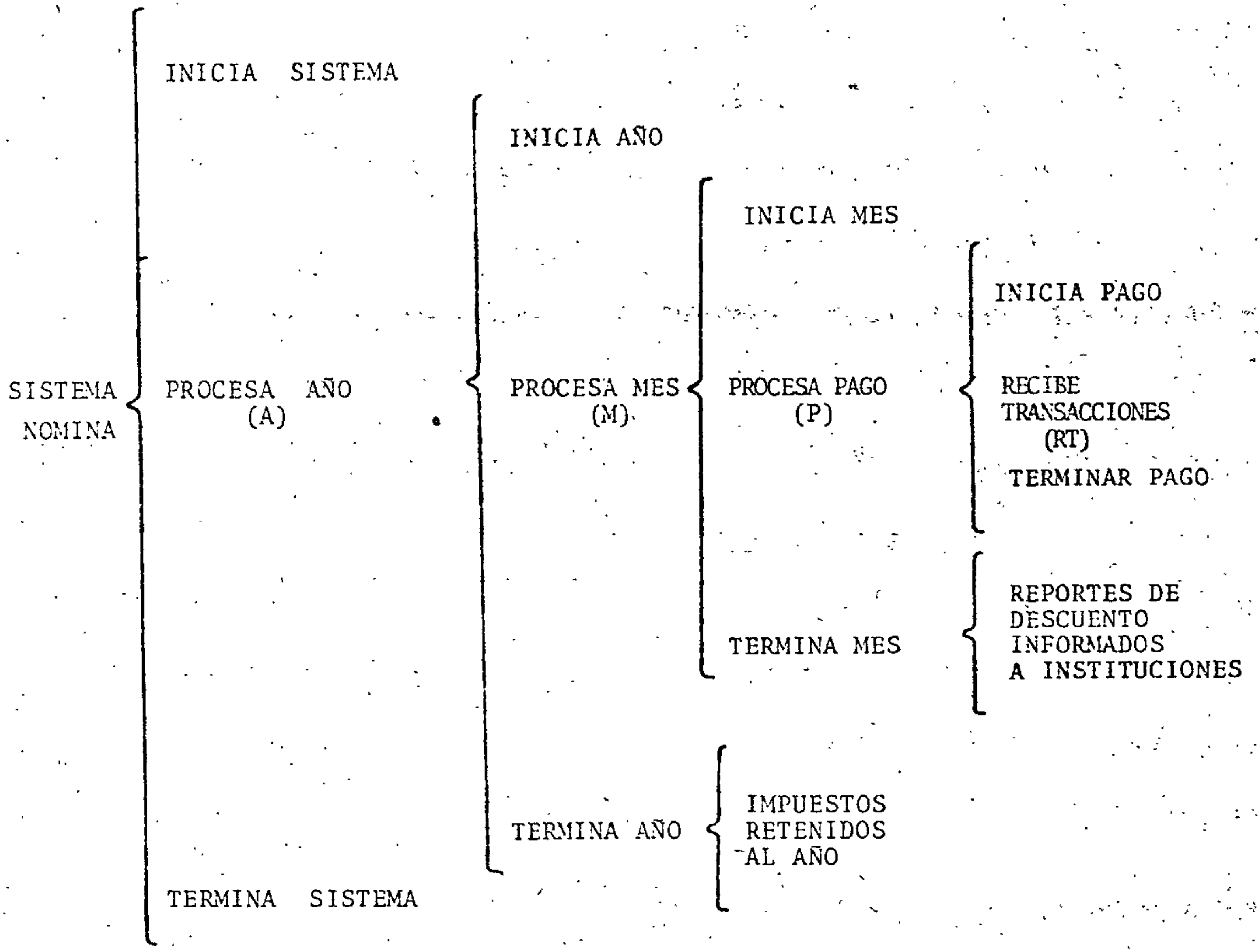
INTEGRA NOMINA EMPLEADOS

⊕

ES OBRERO (0,1)

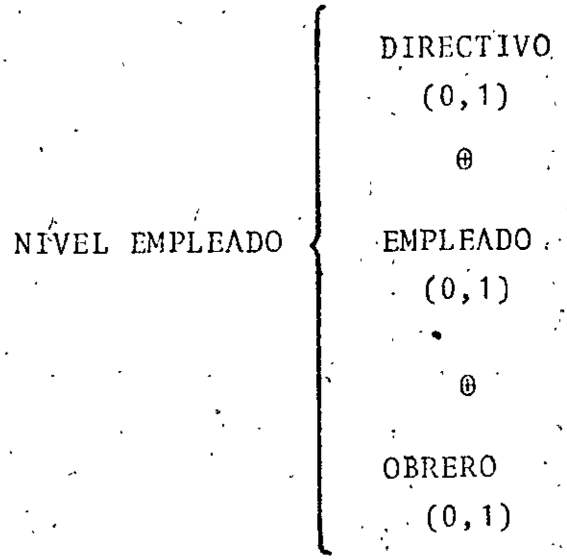
CALCULA OBRERO

INTEGRA NOMINA DE OBREROS



- SISTEMA NOMINA
  - INICIA SISTEMA
  - PROCESA AÑO (A)
    - INICIA AÑO
    - PROCESA MES (M)
      - INICIA MES
      - PROCESA PAGO (P)
        - INICIA PAGO
        - RECIBE TRANSACCIONES (RT)
        - TERMINAR PAGO
      - TERMINA MES
        - REPORTES DE DESCUENTO INFORMADOS A INSTITUCIONES
    - TERMINA AÑO
      - IMPUESTOS RETENIDOS AL AÑO
  - TERMINA SISTEMA

El símbolo (0,1), indica que puede presentarse o no, o bién ocurre cero o una vez, en el caso de datos, la organización de datos nivel empleado se representa por:



NIVEL EMPLEADO

DIRECTIVO (0,1)

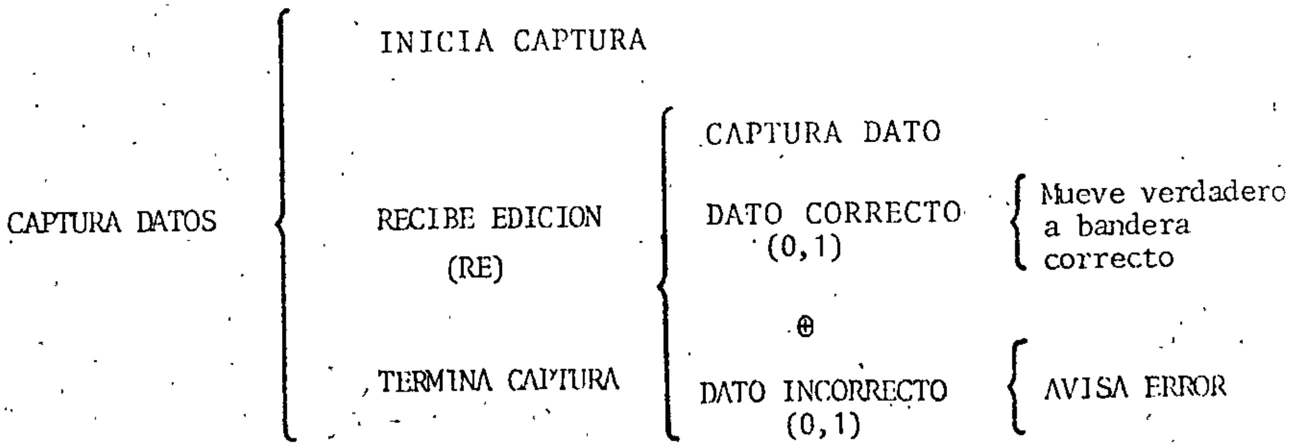
⊕

EMPLEADO (0,1)

⊕

OBRERO (0,1)

Un ejemplo mas de la construcción alternación consiste en editar datos y checar errores.



CAPTURA DATOS

INICIA CAPTURA

RECIBE EDICION (RE)

CAPTURA DATO

DATO CORRECTO (0,1)

Mueve verdadero a bandera correcto

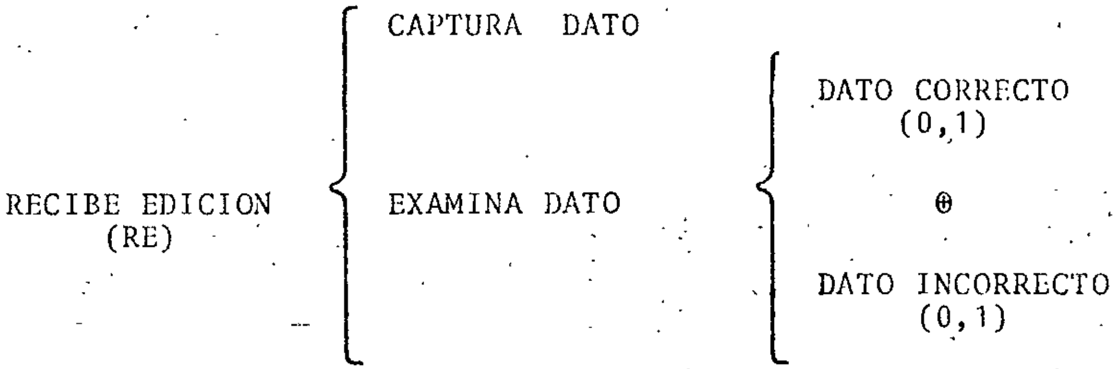
⊕

DATO INCORRECTO (0,1)

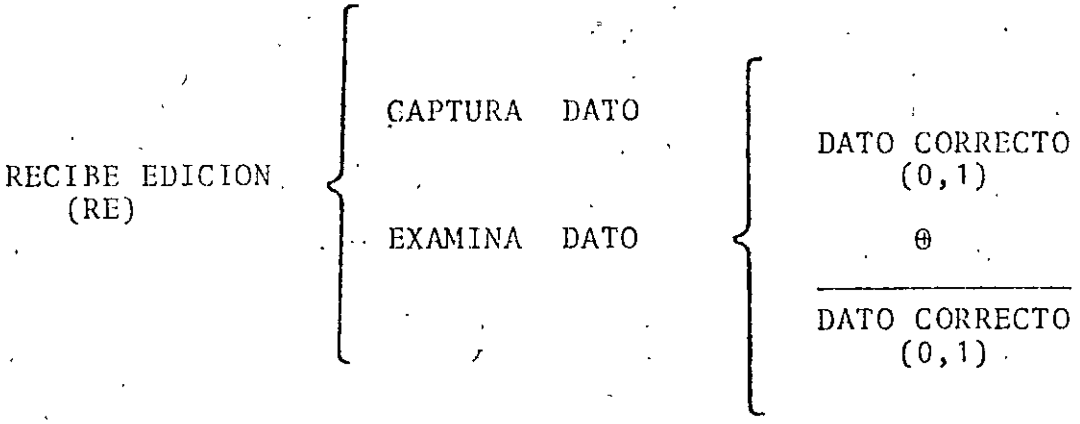
AVISA ERROR

TERMINA CAPTURA

Se puede hacer uso de un símbolo de complemento, si se desea, sin embargo nosotros recomendamos ser explícitos en la expresión por lo que también es necesario agrupar las alternativas en un conjunto adicional y jerárquico como examina dato.



SIN EL USO DE COMPLEMENTO

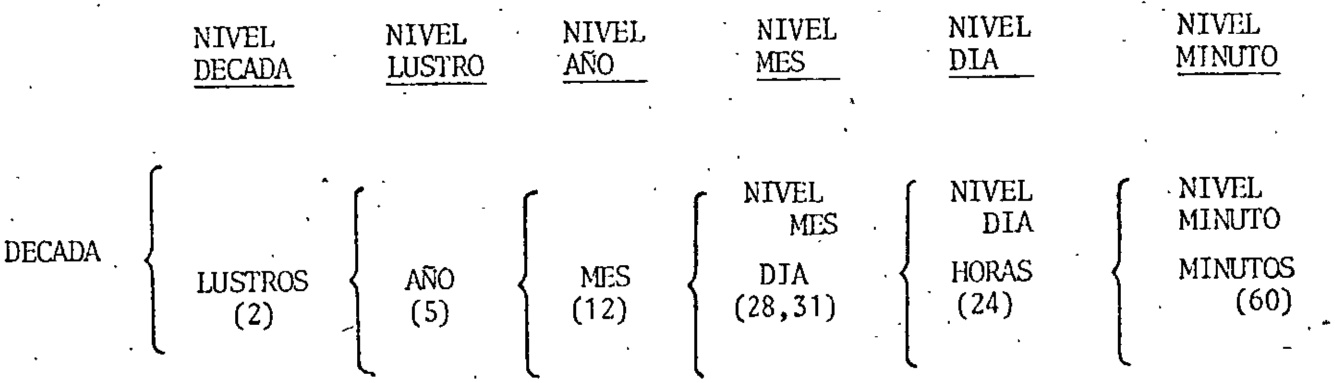


CON EL USO DE COMPLEMENTO

Estructuración basada en construcciones básicas

El concepto utilizado por Ken Orr como estructura se refiere a la utilización combinada de construcciones básicas, dependientes de la solución que estamos representando. Esto es definir jerarquías y/o secuencias de repeticiones y/o conjuntos alternativos, es tan simple la idea, como lo es el -- utilizar pequeños ladrillos en lugar de piedras para realizar grandes proyectos.

Por ejemplo la representación del tiempo partiendo de la década y enumerando los conjuntos jerárquicos y repetitivos.



Con este mismo sentido podríamos continuar enumerando, segúndos, milisegundos hasta parar en un nivel de comprensión de detalle que en un - momento nos lleve a satisfacer el problema.

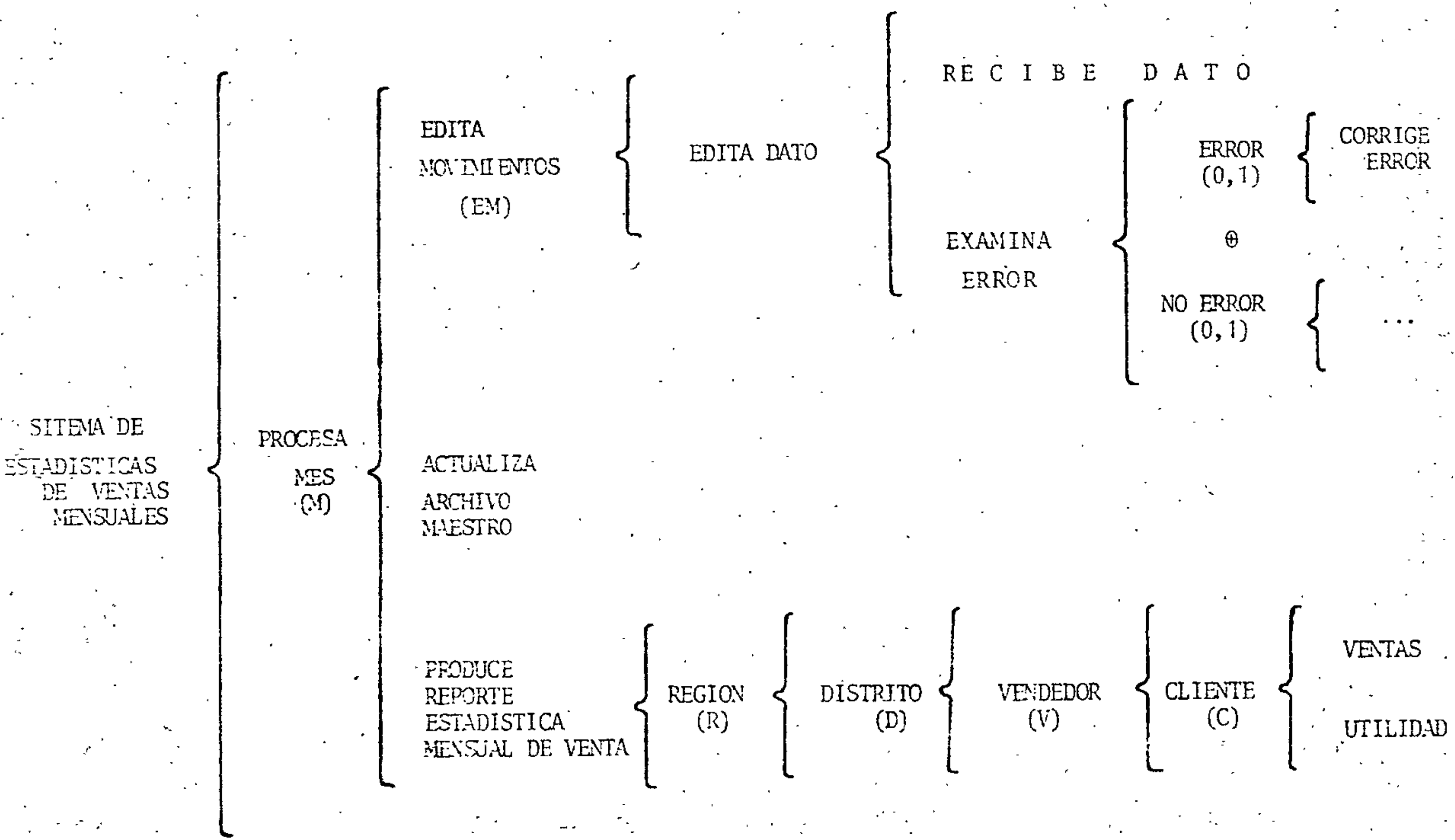
La combinación de construcciones, tiene como objetivos representar:

ESTRUCTURAS DE FUNCIONES/FLUJO DE DATOS

ESTRUCTURAS DE DATOS

ESTRUCTURA DE ACCIONES

Por ejemplo, un sistema de estadísticas de ventas mensuales atravez de estructuras quedaría representado por:


SITEMA DE ESTADISTICAS DE VENTAS MENSUALES
PROCESA MES (M)
EDITA MOVIMIENTOS (EM)
EDITA DATO
RECIBE DATO
EXAMINA ERROR
ERROR (0,1)
CORRIGE ERROR
⊕
NO ERROR (0,1)
...
ACTUALIZA ARCHIVO MAESTRO
PRODUCE REPORTE ESTADISTICA MENSUAL DE VENTA
REGION (R)
DISTRITO (D)
VENDEDOR (V)
CLIENTE (C)
VENTAS
UTILIDAD

En el diagrama mostrado, lo relacionamos inmediatamente a una ubicación en el tiempo y el espacio.

Un sistema de nómina, requiere de calendarizar en el tiempo sus operaciones para efectuar, los pagos e informes de descuentos al gobierno o a las Instituciones, con las que esté relacionado.

El diagrama warnier/orr, es el instrumento para representar como se encuentra actualmente un sistema y como deseamos que esté, para esto identificamos:

- SALIDAS
- PERIODICIDAD
- ENTRADAS POSIBLES
- PROCESOS MANUALES
- PROCESOS AUTOMATIZADOS
- BASE DE DATOS
- CONTROLES

El proceso de estructuración, es un arte basado en pequeñas construcciones que nos llevan a la solución de <u>Sistemas Correctos y Civilizados.</u> En lugar de enfrentarnos con un monolito indestructible, lo hacemos con las herramientas adecuadas, las pensamos y éstas finalmente se aplican a la solución del problema.

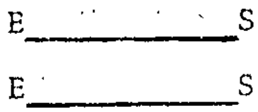
MAPEOS

La filosofía básica del método WARNIER consiste en que los problemas informáticos son una extensión de las matemáticas denominada teoría de conjuntos.

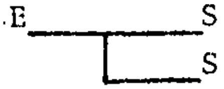
Un conjunto es una colección relacionada de cosas. En este sentido, toda colección de datos que guardamos y trabajamos son conjuntos, y todos los programas son básicamente una -- transformación básada en reglas de diseño. Para pasar de - un conjunto a otro. A dicha transformación la denominamos MAPEO (REGLA DE ASOCIACION).

Existen tres clases de mapeos relativamente simples, de entradas (E) a salidas (S).

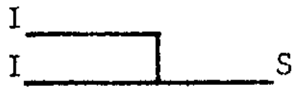
1) UNO A UNO (identidad)



2) UNO A MUCHOS (relación)

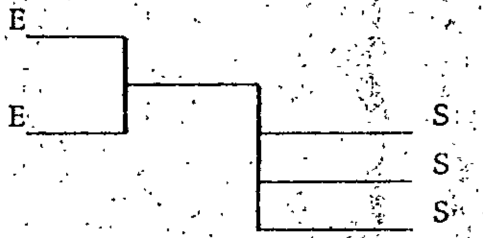


3) MUCHOS A UNO (función)



Cuando transformamos entradas en salidas relativamente si aplicamos los anteriores mapeos, necesariamente mejoraríamos nuestro diseño, sin embargo, la mayoría de programas interesantes entran en la categoría del mapeo complejo.

MUCHOS A MUCHOS (complejo)



Como se indica en la figura, varios registros de entrada son coleccionados y usados para generar varios registros de salida. En la práctica este tipo de mapeos son excesivamente difíciles de detallar dando como consecuencia razonamientos y soluciones oscuras.

En informática como en muchas disciplinas existentes como -- contrario a lo complejo buscamos lo simple, de esto nace la mejor ingeniería.

ESTRUCTURAS LOGICAS DE SALIDAS (ELS)

La estructura lógica de salida (ELS) es el diagrama WARNIER/ ORR, mediante el cual se describe detalladamente los requerimientos de salida y el diseño lógico correcto para representar una salida.

Básicamente esto cumple con dos propósitos:

1) Es una excelente herramienta para adquirid un conocimiento detallado de los datos y para verificar el entendimiento con el usuario.

2) Mediante este instrumento adquirimos la información mas relevante para estructurar los datos lógicos. Necesarios para las estructuras lógicas de proceso.

El término estructura, como mencionamos es; una jerarquía y/o secuencia de repeticiones y/o conjuntos alternativos (o jerarquías).

Generalmente la presentación tradicional de una salida se hace a través de reportes impresos, esto se debe a simple conveniencia. Existen varias formas en que las salidas se presentan, tratando de enumerar sólo algunas mencionaremos: cinta magnética, tarjetas perforadas, microfichas, pantallas, discos magnéticos etc.

Tomaremos como ejemplo algunas salidas en forma de reporte, para explicar la representación estructurada de acuerdo a las construcciones básicas.

La representación mas sencilla es la estructura lógica de salidas en forma secuencial, como mencionamos, esta se identifica porque los datos se encuentran sucesivos, supongamos la representación de una etiqueta para enviar comunicación a clientes:

NOMBRE CLIENTE

DIRECCION

CIUDAD

C.P.

La estructura lógica de salida, para representar la etiqueta sería:

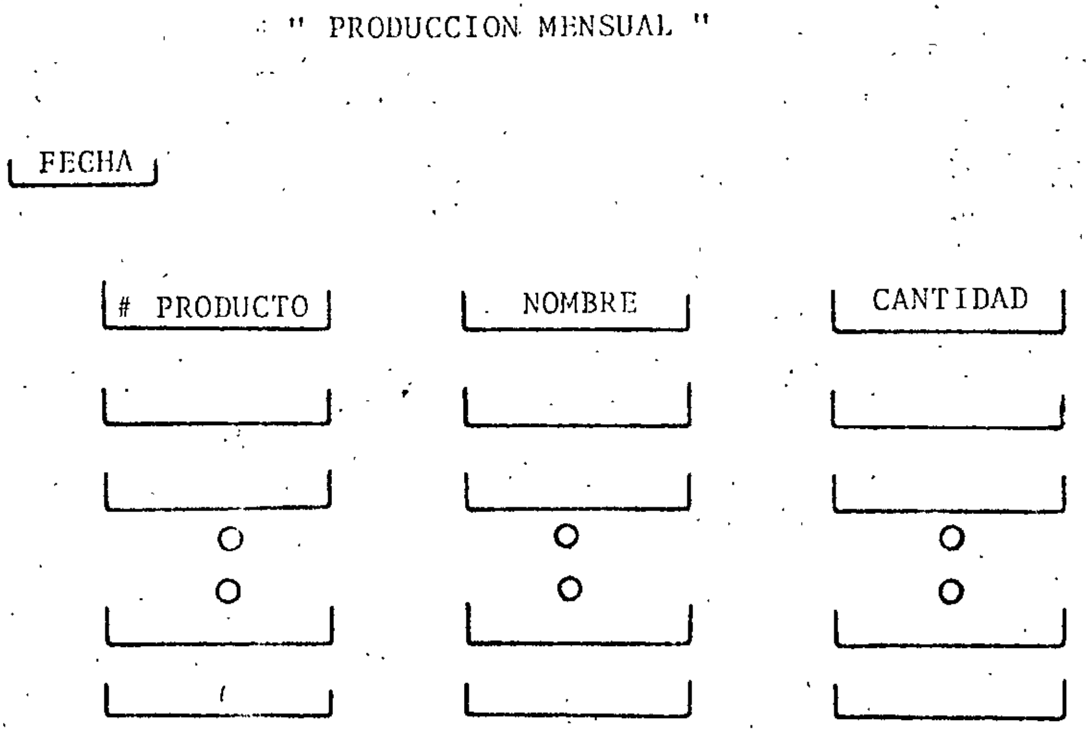
$$\text{ETIQUETA}\left\{\begin{array}{l}\text{NOMBRE CLIENTE}\\ \text{DIRECCION}\\ \text{CIUDAD}\\ \text{C. P.}\end{array}\right.$$

Donde se lee, el conjunto etiqueta <u>CONTIENE A</u>, nombre cliente, dirección, ciudad, C. P.

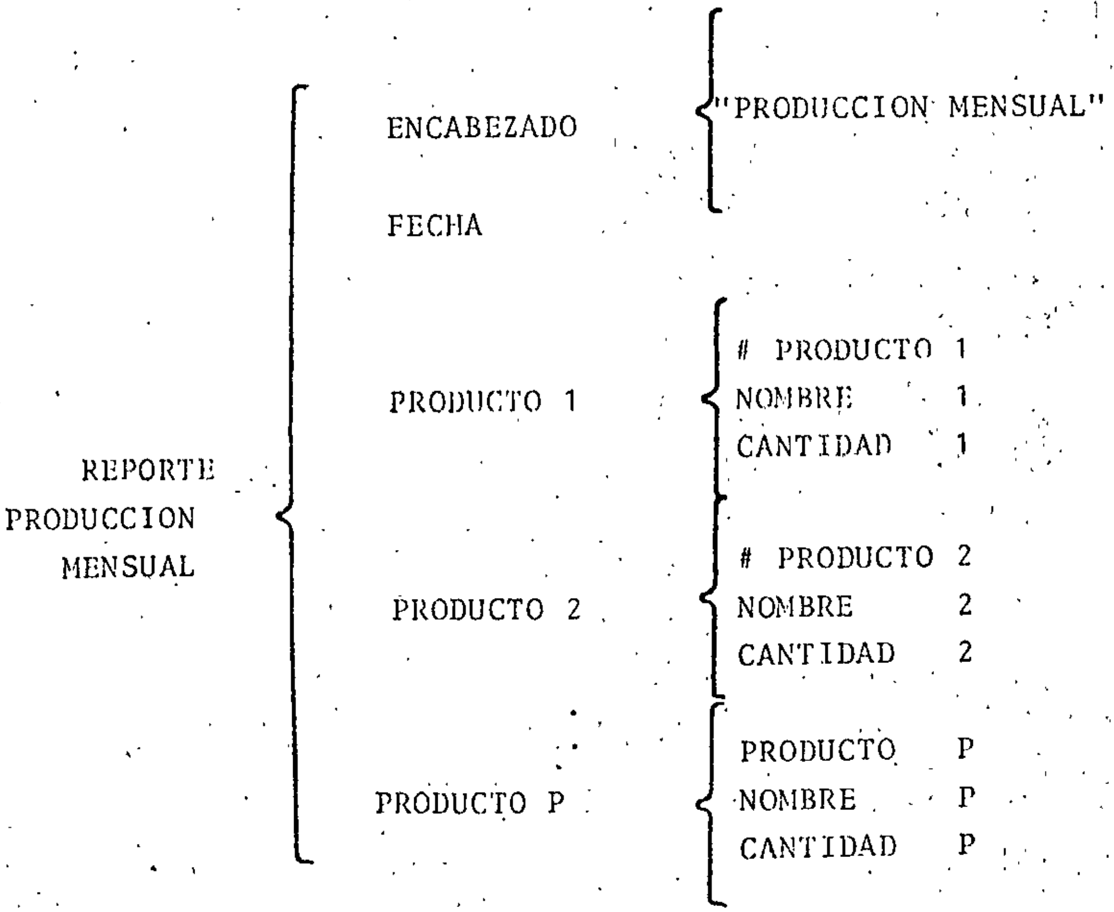
Cuando existe mas de una etiqueta como el anterior ejemplo, lo conveniente es utilizar una construcción repetitiva.

$$\begin{array}{c}\text{ETIQUETA}\\ \text{(E)}\end{array}\left\{\begin{array}{l}\text{NOMBRE CLIENTE}\\ \text{DIRECCION}\\ \text{CIUDAD}\\ \text{C.P.}\end{array}\right.$$

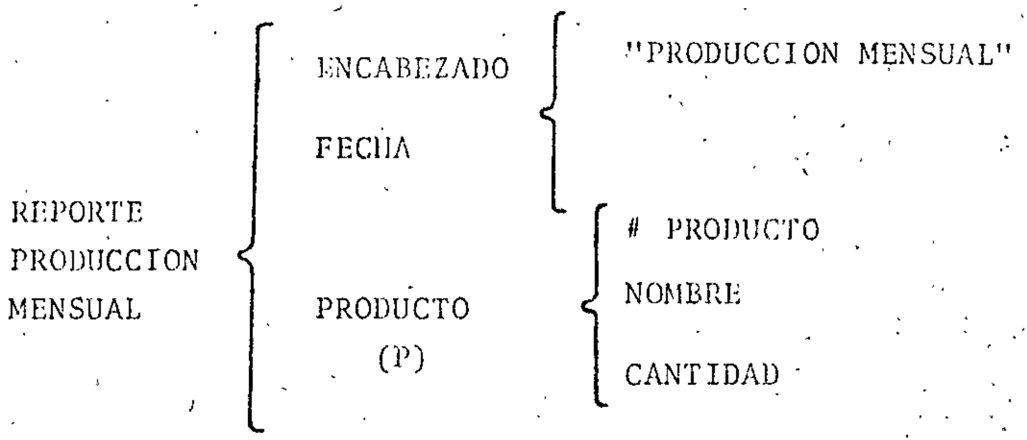
Donde se lee, el conjunto etiqueta sucede E veces y cada etiqueta <u>CONTIENE A,</u> nombre cliente, dirección, ciudad y C. P.
Un ejemplo mas de como se relacionan las construcciones secuenciales y repetitivas, se muestra mediante la representación de un reporte de producción mensual de los productos hechos por una compañía X.



Para pasarlo a un diagrama WARNIER/ORR identificamos las partes como lo ordenan las secuencias

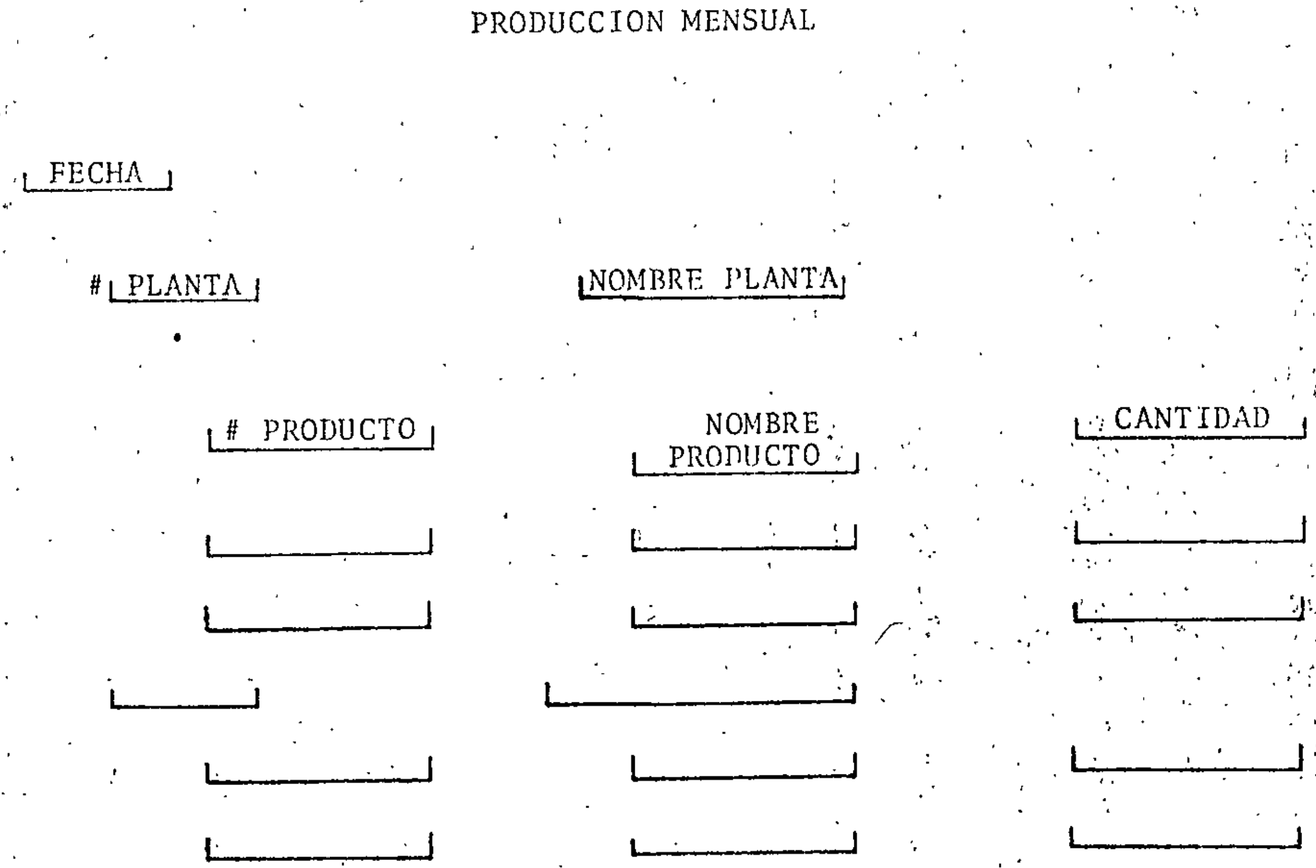


Con respecto a las secuencias, vemos que los productos suceden varias veces, es decir, las características se repiten, en este caso lo mejor es representarlo mediante una construcción básica repetitiva, quedando el diagrama como:

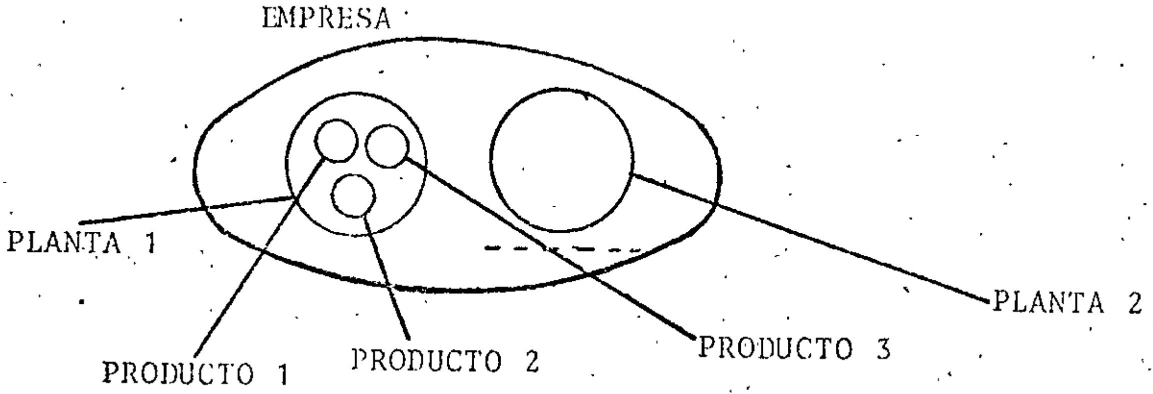


La repetición muestra que un producto sucede P veces, después de encabezados y fecha.

Ahora bien si el problema se trasladara a representar la estructura lógica de un reporte mensual de producción, donde se reportan los productos por planta productora, fisicamente el reporte cambiaría de forma y por lo tanto la estructura lógica también. por ejemplo:



La estructura lógica tendría la siguiente representación, pensando en conjuntos:

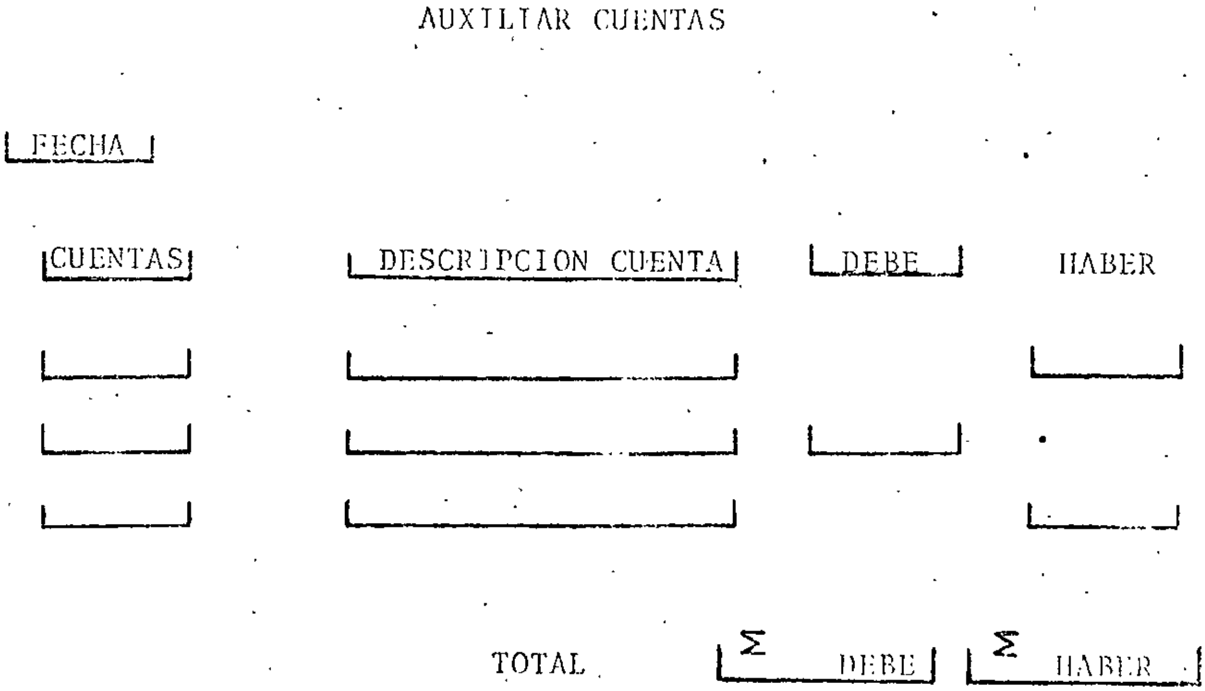


EMPRESA { PLANTA (P) { PRODUCTOS (P)

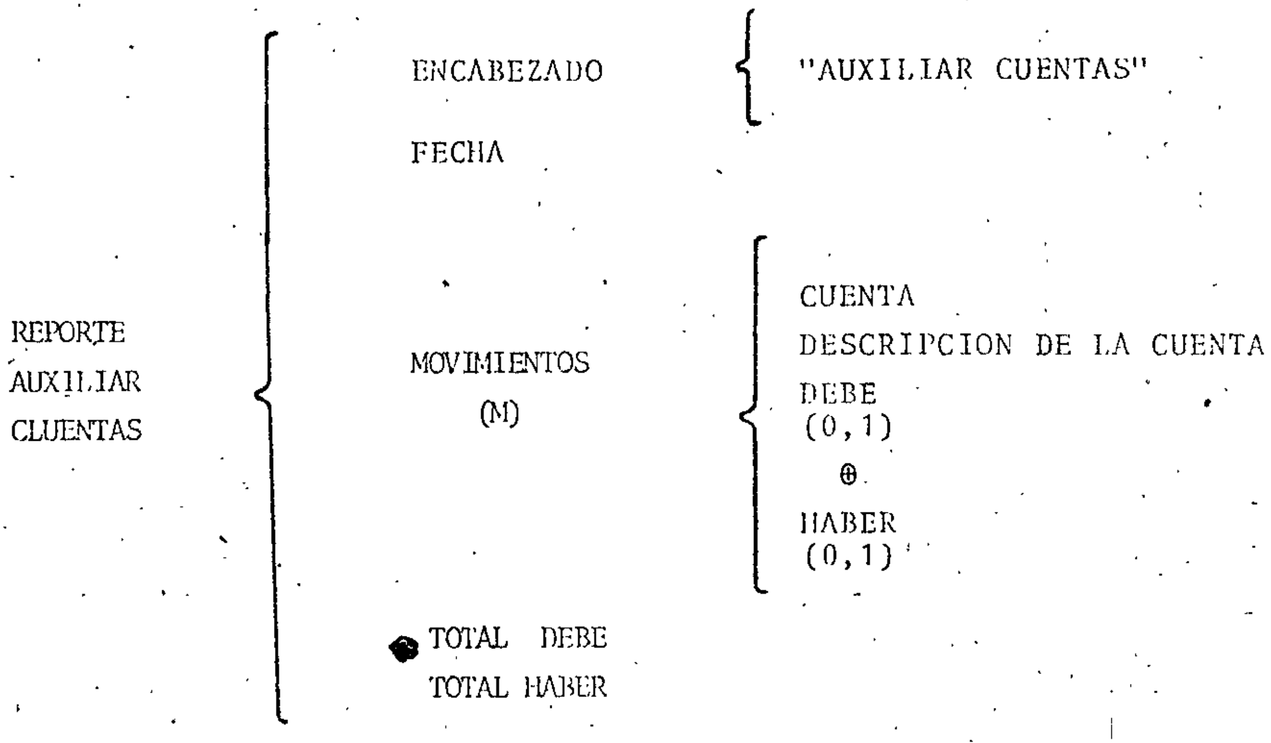
La estructura lógica de salida, siguiendo éste razonamiento sería:

REPORTE MENSUAL PRODUCCION DE PLANTAS {
- ENCABEZADO { "PRODUCCION MENSUAL
- FECHA
- PLANTA (P) {
  - # PLANTA
  - NOMBRE PLANTA
  - PRODUCTO (PR) {
    - # PRODUCTO
    - NOMBRE
    - CANTIDAD

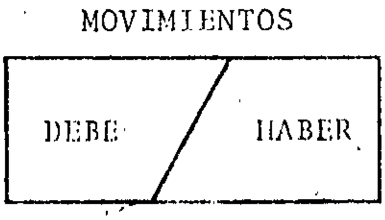
Un reporte típico de una empresa es el Auxiliar Cuentas, este ejemplo presenta la aplicación de la construcción alternativa, también denominada conjunto complemento.



La estructura lógica de salida del Auxiliar se representa mediante el siguiente diagrama:



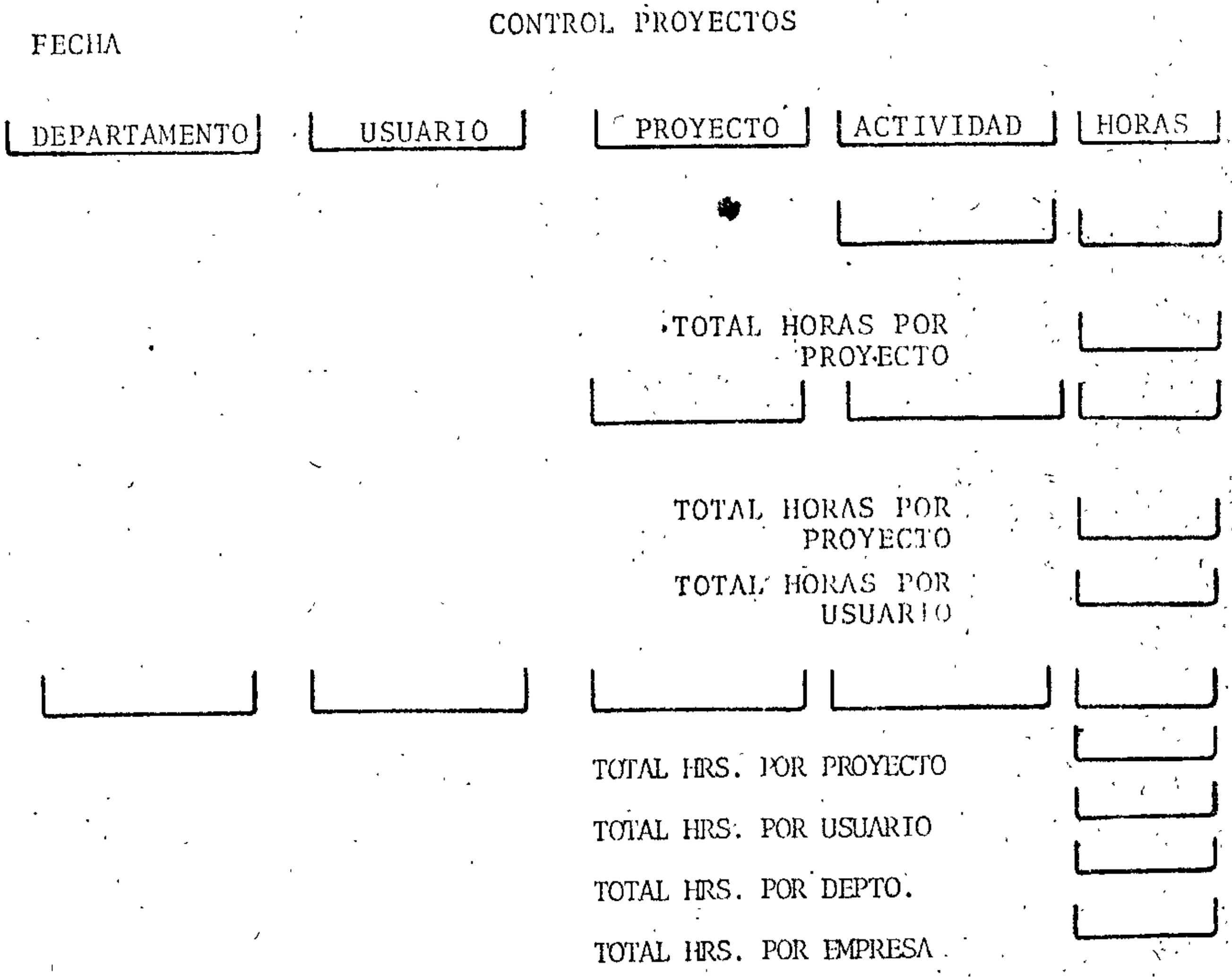
Nosotros observamos la aplicación del conjunto alternativo al monto asignado a la cuenta; ya sea en el Debe o en el Haber en forma exclusiva.



En el diagrama de Venn se clarifica el concepto de conjunto complementario, y donde además se nota el concepto de exclusión.

La estructuración del "REPORTE MAYOR", muestra la combinación de construcciones básicas jerarquías y/o secuencias y/o repeticiones y/o alternaciones.

Una estructura lógica de salida aplicada a control de proyectos, con un gran número de jerarquías, para acumular horas -- por proyecto en un departamento, y para acumúlar horas asignadas a un usuario del proyecto en un departamento, tipicamente puede ser:



CONTROL PROYECTOS

FECHA

DEPARTAMENTO USUARIO PROYECTO ACTIVIDAD HORAS

TOTAL HORAS POR PROYECTO

TOTAL HORAS POR PROYECTO

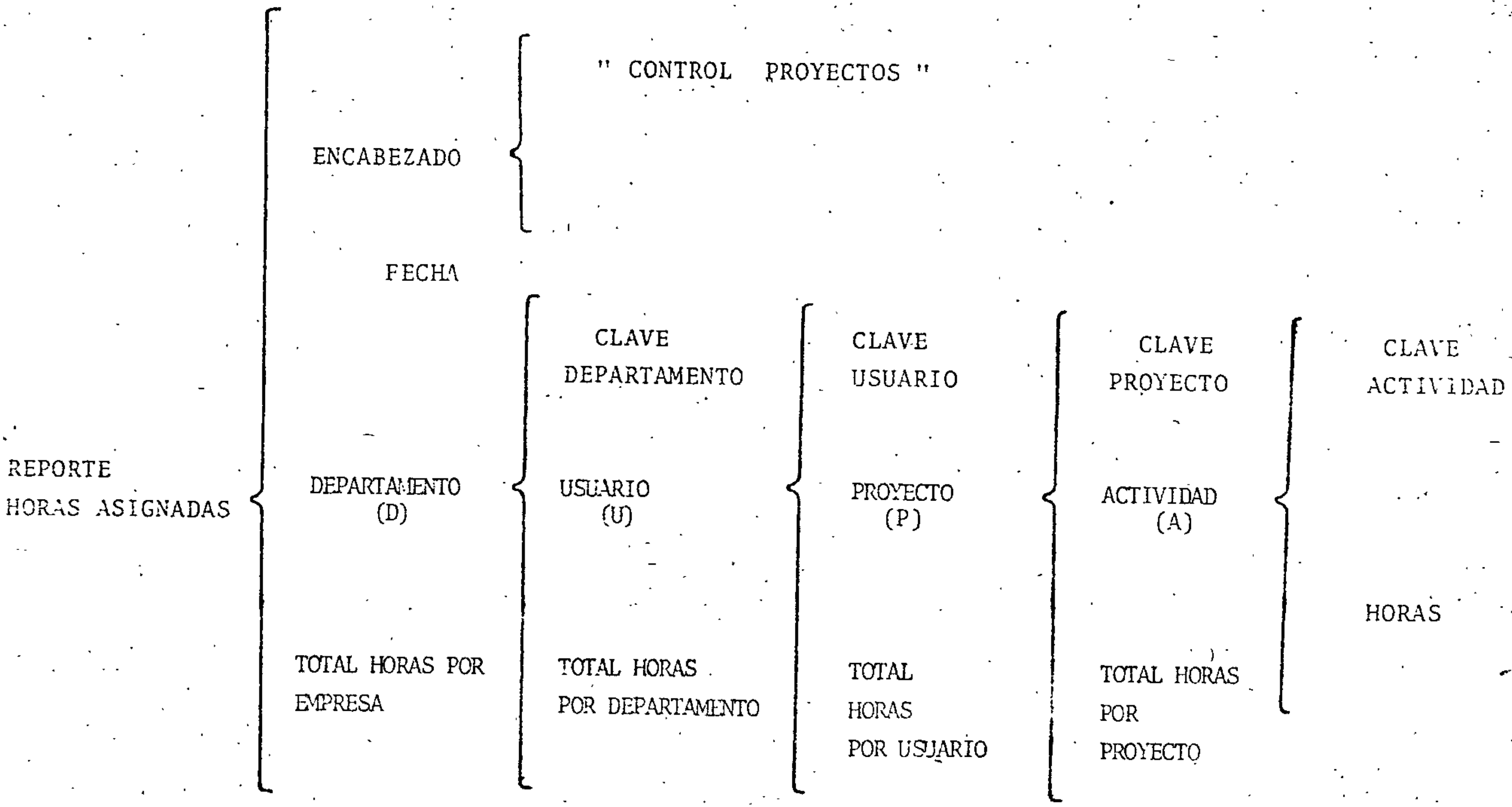
TOTAL HORAS POR USUARIO

TOTAL HRS. POR PROYECTO

TOTAL HRS. POR USUARIO

TOTAL HRS. POR DEPTO.

TOTAL HRS. POR EMPRESA



REPORTE HORAS ASIGNADAS

ENCABEZADO

" CONTROL PROYECTOS "

FECHA

DEPARTAMENTO (D)

CLAVE DEPARTAMENTO

USUARIO (U)

CLAVE USUARIO

PROYECTO (P)

CLAVE PROYECTO

ACTIVIDAD (A)

CLAVE ACTIVIDAD

HORAS

TOTAL HORAS POR PROYECTO

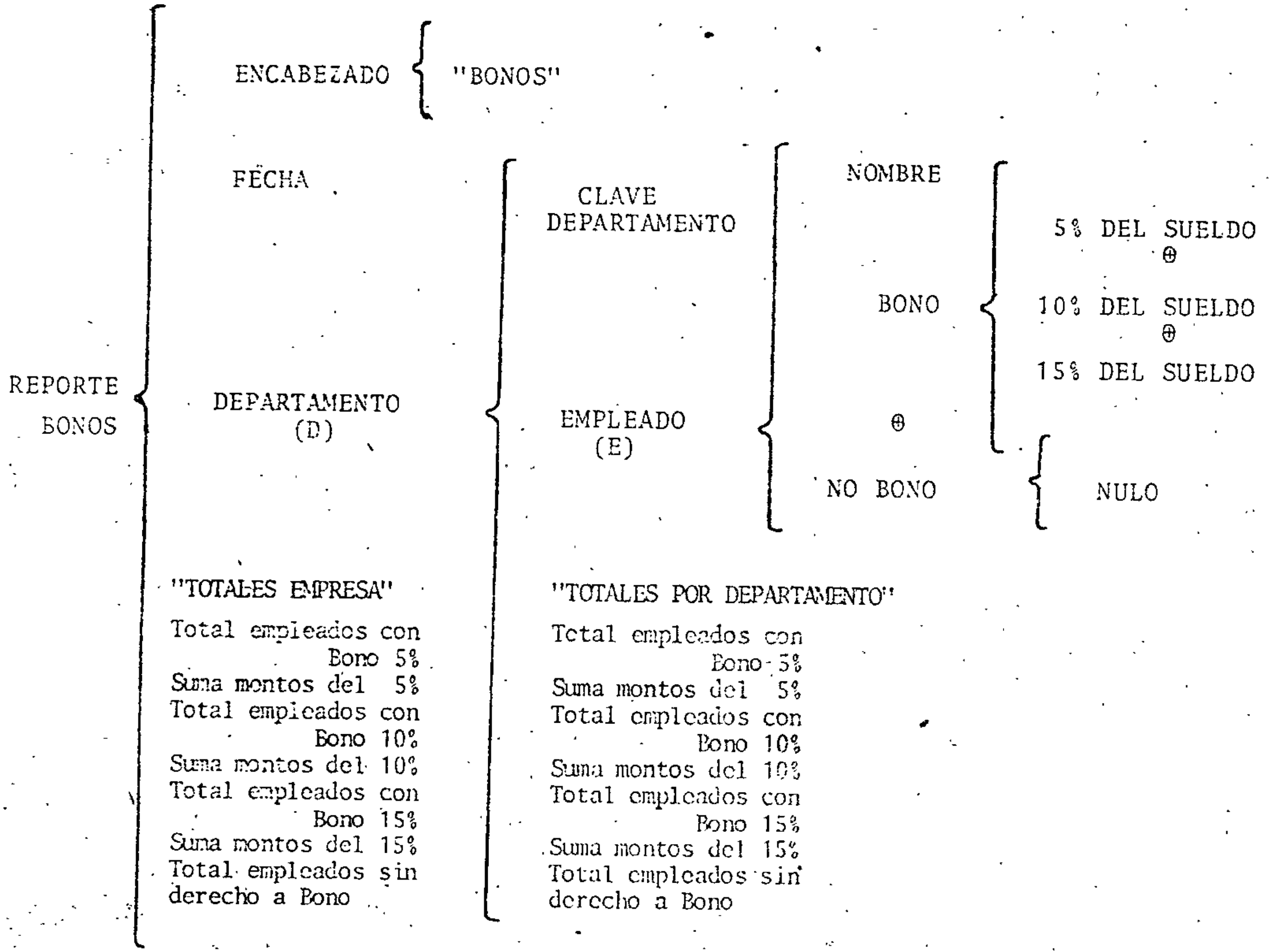
TOTAL HORAS POR USUARIO

TOTAL HORAS POR DEPARTAMENTO

TOTAL HORAS POR EMPRESA

La estructura lógica de salida puede mostrar estructuras tan complejas que en un reporte típico sería díficil mostrar o requeriría de una narrativa especial que evitaría la claridad y en ocasiones provocaría el error.

Un ejemplo de este caso consiste en asignar un bono a los empleados, que puede consistir del 5%, 10% ó 15% con base al sueldo, o bien nada si no lo ameritan, este hecho pudiera representarse mediante el diagrama siguiente:



REPORTE BONOS

ENCABEZADO

"BONOS"

FECHA

DEPARTAMENTO (D)

CLAVE DEPARTAMENTO

EMPLEADO (E)

NOMBRE

BONO

5% DEL SUELDO

⊕

10% DEL SUELDO

⊕

15% DEL SUELDO

⊕

NO BONO

NULO

"TOTALES POR DEPARTAMENTO"

Total empleados con Bono 5%
Suma montos del 5%
Total empleados con Bono 10%
Suma montos del 10%
Total empleados con Bono 15%
Suma montos del 15%
Total empleados sin derecho a Bono

"TOTALES EMPRESA"

Total empleados con Bono 5%
Suma montos del 5%
Total empleados con Bono 10%
Suma montos del 10%
Total empleados con Bono 15%
Suma montos del 15%
Total empleados sin derecho a Bono

En el diagrama anterior, se muestra la necesidad de producir el reporte por departamento, conteniendo a los empleados correspondientes, con el requerimiento previamente descrito, también notamos que en el reporte de bonos se acumulan totales por empresa y por departamento.

La estructuración de las salidas, tiene un impacto en la definición de requerimientos y en el diseño lógico, de -- tal magnitud que cualquier cambio en las salidas tiene como consecuencia una afectación profunda en la arquitectura del sistema.

Como una síntesis importante para resumir el concepto de estructuras lógicas de salida, definiremos el procedimiento para hacer explícitos los requerimientos de salida.

1) DEFINICION DE DATOS ELEMENTALES:

Esto se hace mediante una lista de los datos que aparecen en el reporte de salida.

2) DEFINICION DE FRECUENCIAS:

Este paso consiste en descubrir la frecuencia de cada elemento que aparece en el reporte de salida (identificación de repeticiones y jerarquías)

3) DEFINICION DE EXCLUSIONES:

El objetivo de este paso es investigar, mediante el usuario o en narrativas existentes cuales son las alternaciones presentadas, así como su fuente.

4) MAPEAR A LA ESTRUCTURA:

Esto consiste en mapear las frecuencias y las exclusiones definidas, en los diagramas Warnier/orr, de acuerdo a la notación aplicada a las construcciones básicas.

5) PROBAR LOGICAMENTE LA ESTRUCTURA LOGICA DE SALIDA:

Esto consiste en revisar, y observar la estructura con el objetivo de encontrar errores y corregirlos.

Puedo concluir después de lo anterior que los diagramas Warnier no solo expresan simples algoritmos, ayudan a hacer simples algoritmos y recordemos que solo lo sencillo puede minimizar lo complejo.

ESTRUCTURAS LOGICAS DE DATOS (ELD)

Las estructuras lógicas de datos tienen importancia después de la definición de las estructuras lógicas de salida (ELS), ya - que a partir de esto su derivación se vuelve relativamente fácil, una definición sencilla de lo que significan las (ELD) es:

El conjunto mínimo de datos de entrada requeridos para producir la salida, de donde además, se busca una utilización para una o más (ELS), y que optimice logicamente las funciones de proceso de datos tales como; codificación, validación, actualización y recuperación.

Los pasos mas importantes para estructurar las (ELD) son:

1) Hacer una lista de los datos elementales de la (ELS) con los niveles que tienen.

2) Identificar los datos de la (ELS) que sean constantes o simples etiquetas.

3) Identificar los datos calculados describiendo las reglas.

4) Identificar los datos primarios, simplificando las alternaciones mediante un código de identificación.

5) Hacer un mapeo de la jerarquía de la (ELS) a la (ELD).

6) Asignar los datos primarios.

7) Depositar los datos primarios en sus correspondientes jerarquías.

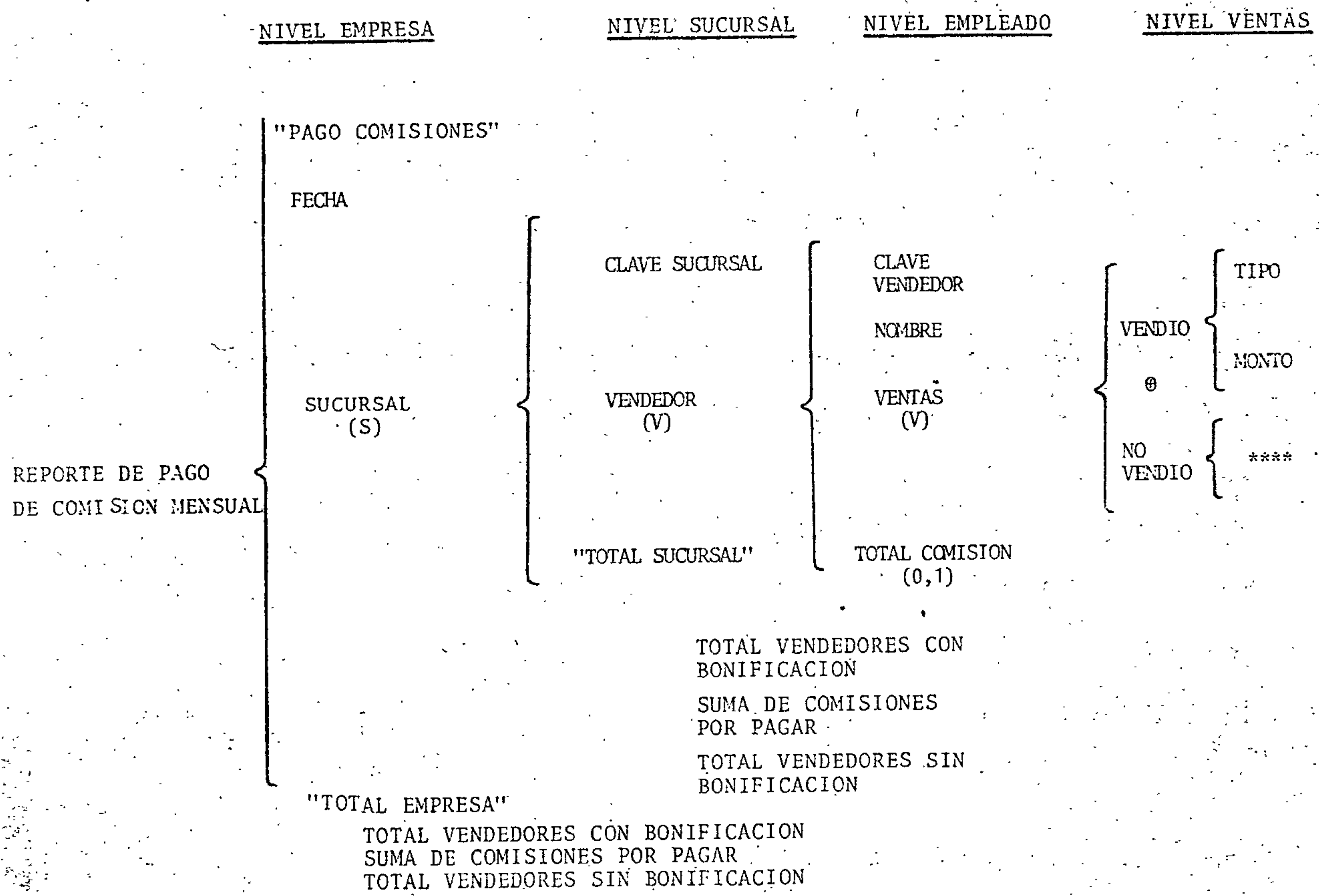
8) Proyectar los datos primarios de las jerarquías izquierdas hasta la última derecha.

9) Revisar y probar logicamente la estructura lógica de datos.

Para ilustrar de manera mas precisa el procedimiento, utilizaremos una (ELS), básada en un problema de ventas, donde el objetivo es pagarle una comisión a los vendedores existentes en las diferentes sucursales a partir de los productos facturados y el tipo de venta, de contado, y crédito especial o bien, 30, 60, 90 días, esto implica que en relación a esto se respete una regla de correspondencia dada por la siguiente tabla:

VENTA	COMISION	TIPO
CONTADO	20%	A
30 DIAS	18%	B
60 DIAS	15%	C
90 DIAS	11%	D
CR. ESPECIAL	06%	E

Se desean totales por vendedores y una proyección de lo que se paga por comisión en las sucursales y a nivel de la empresa, se debe indicar qué vendedores tienen y quienes no tienen la comisión.



NIVEL EMPRESA

NIVEL SUCURSAL

NIVEL EMPLEADO

NIVEL VENTAS

REPORTE DE PAGO DE COMISION MENSUAL

"PAGO COMISIONES"

FECHA

SUCURSAL (S)

CLAVE SUCURSAL

VENDEDOR (V)

CLAVE VENDEDOR

NOMBRE

VENTAS (V)

VENDIO

TIPO

MONTO

⊕

NO VENDIO

****

TOTAL COMISION (0,1)

"TOTAL SUCURSAL"

TOTAL VENDEDORES CON BONIFICACION

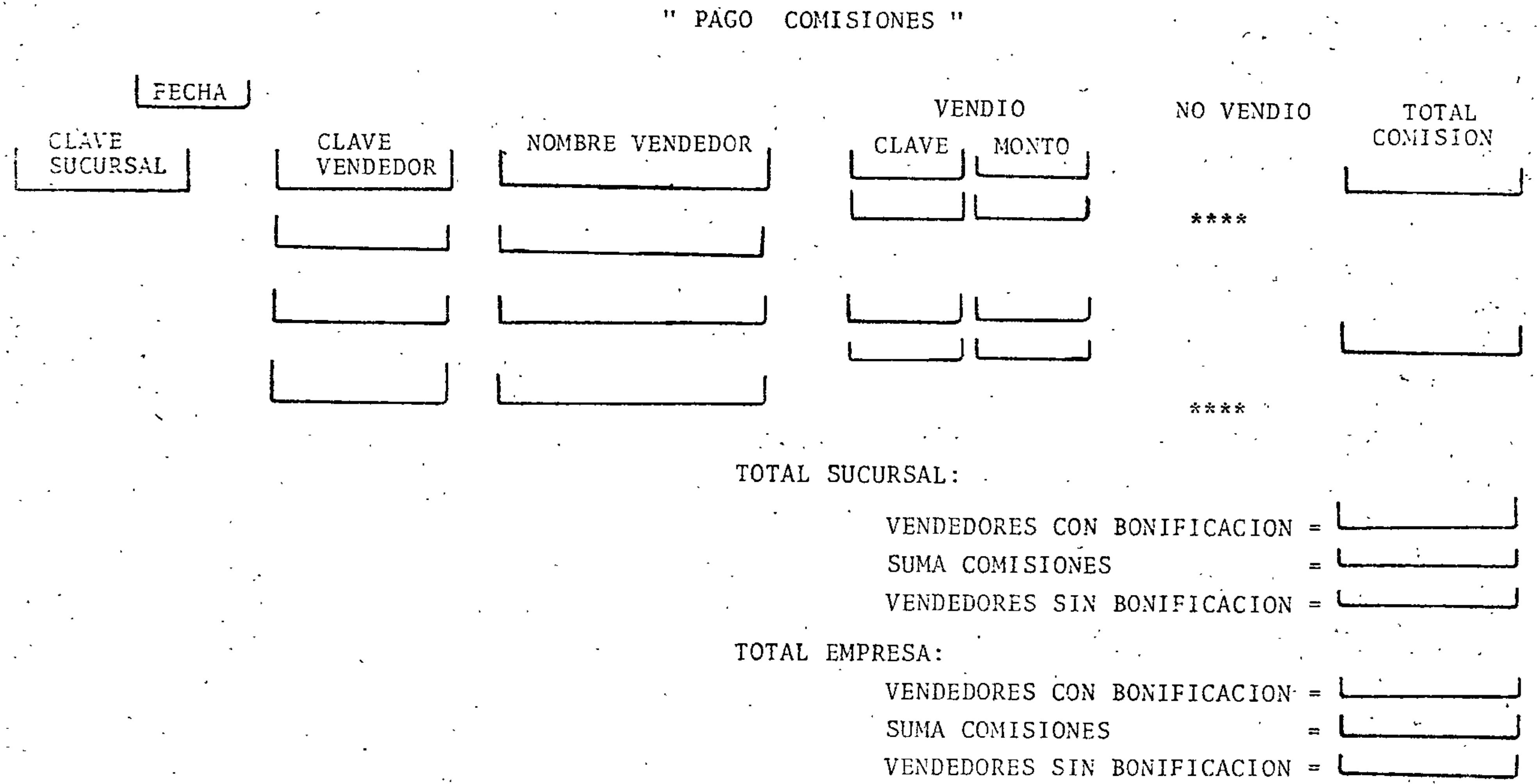
SUMA DE COMISIONES POR PAGAR

TOTAL VENDEDORES SIN BONIFICACION

"TOTAL EMPRESA"

TOTAL VENDEDORES CON BONIFICACION
SUMA DE COMISIONES POR PAGAR
TOTAL VENDEDORES SIN BONIFICACION

FISICAMENTE EL REPORTE PUDIERA SER:



" PAGO COMISIONES "

FECHA

CLAVE SUCURSAL

CLAVE VENDEDOR

NOMBRE VENDEDOR

VENDIO

CLAVE

MONTO

NO VENDIO

TOTAL COMISION

****

****

TOTAL SUCURSAL:

VENDEDORES CON BONIFICACION =

SUMA COMISIONES =

VENDEDORES SIN BONIFICACION =

TOTAL EMPRESA:

VENDEDORES CON BONIFICACION =

SUMA COMISIONES =

VENDEDORES SIN BONIFICACION =

De acuerdo a nuestro programa y a los pasos descritos con anterioridad, realizamos los siguientes ejercicios:

PASO 1

LA LISTA DE DATOS INICIAL SERIA:

NUM	DATO	NIVEL
1	"PAGO COMISIONES"	EMPRESA
2	FECHA	EMPRESA
3	CLAVE SUCURSAL	SUCURSAL
4	CLAVE VENDEDOR	VENDEDOR
5	NOMBRE	VENDEDOR
6	TIPO	VENTAS
7	MONTO	VENTAS
8	"*****"	VENTAS
9	TOTAL COMISION	VENDEDOR
10	"TOTAL SUCURSAL	SUCURSAL
11	VENDEDORES C/BON	SUCURSAL
12	SUMA BONIFICACIONES	SUCURSAL
13	VENDEDORES S/BON	SUCURSAL
14	"TOTAL EMPRESA"	EMPRESA
15	"VENDED.CON BONIFICACION"	EMPRESA
16	SUMA BONIFICACIONES	EMPRESA
17	VENDEDORES SIN BONIF.	EMPRESA

Señalamos a continuación:

PASO 2

Los datos que aparecen como constantes

seleccionamos

1, 8, 10,14

PASO 3

Indicamos las reglas de cálculo para datos derivados.

NUMERO DATO	REGLA
9	SI EXISTEN COMISIONES SE SUMAN POR VENDEDOR.
11	SE SUMA 1 POR CADA VENDEDOR QUE AL MENOS HAYA TENIDO UNA VENTA.
12	SE SUMA EL MONTO TOTAL DE LA COMISION, A ESTE CAMPO.
13	SE SUMA 1 POR CADA VENDEDOR QUE NO HAYA TENIDO VENTAS.
15	SE SUMA EL TOTAL DE VENDEDORES CON BONIFICACION DE CADA SUCURSAL.
16	SE SUMA EL MONTO TOTAL DE PAGO DE COMISION DE CADA SUCURSAL.
17	SE SUMA EL TOTAL DE VENDEDORES SIN BONIFICACION DE CADA EMPRESA.

PASO 4

Los datos primarios que quedan son:

NUMERO	DATO ELEMENTAL
2	FECHA
3	CLAVE SUCURSAL
4	CLAVE VENDEDOR
5	NOMBRE
6	TIPO COMISION
7	MONTO COMISION

PASO 5

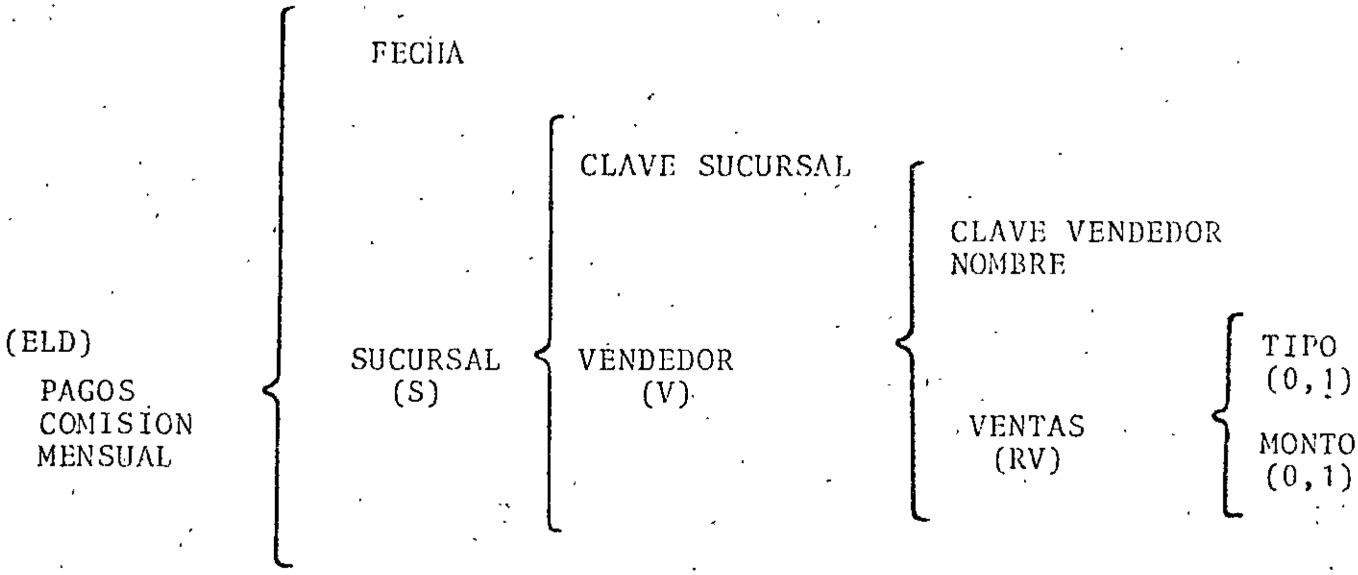
Para el caso donde no exista comisión:

EL DATO 6, "TIPO" y el
DATO 7, "MONTO", DEBERAN CONTENER CEROS.

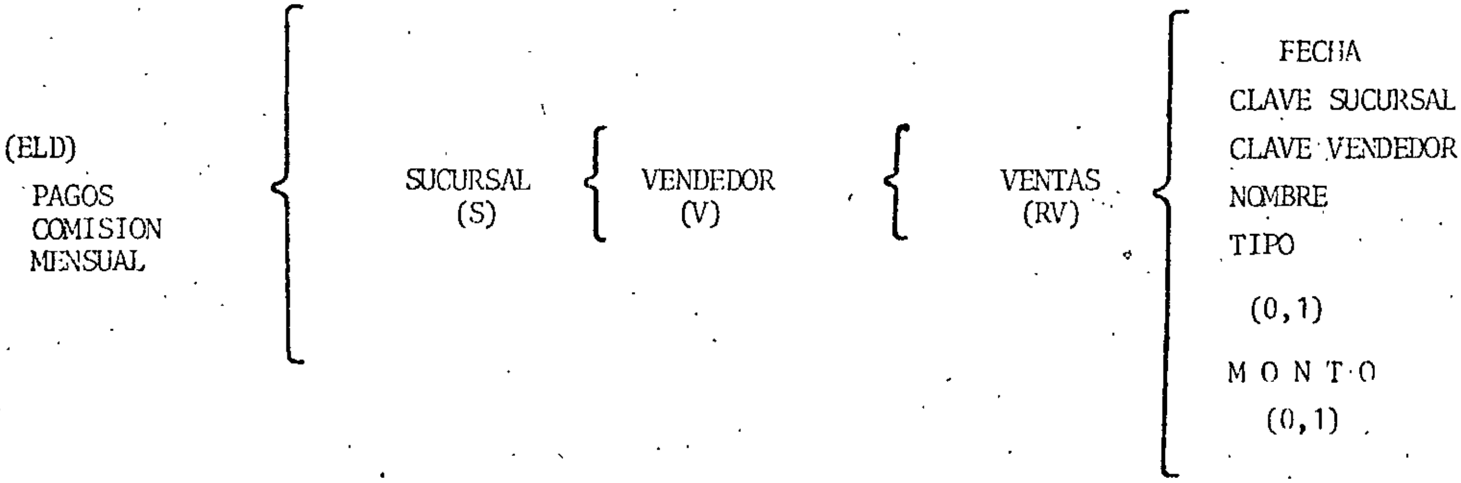
PASO 6

DATOS LOGICOS PAGOS COMISION MENSUAL { SUCURSAL (S) { VENDEDOR (V) { VENTAS (RV)

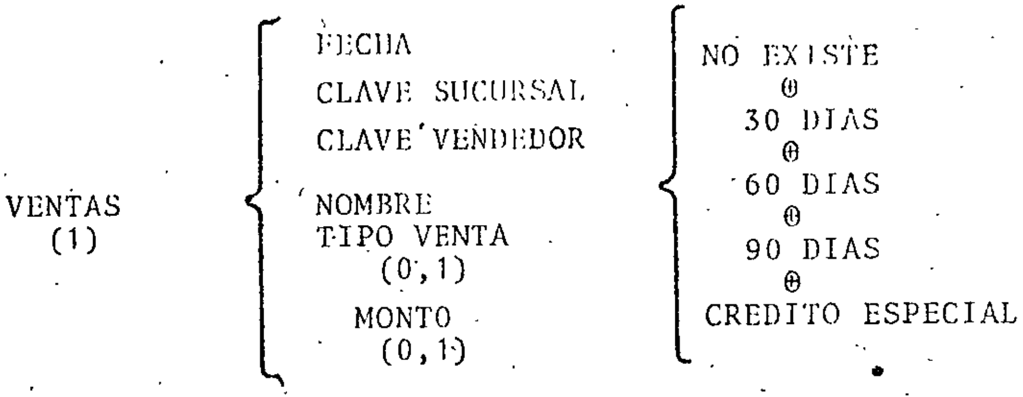
PASO 7


(ELD)
PAGOS
COMISION
MENSUAL
FECHA
SUCURSAL
(S)
CLAVE SUCURSAL
VENDEDOR
(V)
CLAVE VENDEDOR
NOMBRE
VENTAS
(RV)
TIPO
(0,1)
MONTO
(0,1)

PASO 8


(ELD)
PAGOS
COMISION
MENSUAL
SUCURSAL
(S)
VENDEDOR
(V)
VENTAS
(RV)
FECHA
CLAVE SUCURSAL
CLAVE VENDEDOR
NOMBRE
TIPO
(0,1)
MONTO
(0,1)

Un paso final que define la estructura lógica de datos, es representarla de manera que solo sea necesario agregarle longitudes y clases (numérico, alfanumérico, etc) a los datos y a partir de esto requerir la codificación y captura al usuario para obtener y procesar la estructura lógica de salida.



BASE DE DATOS LOGICA (BDL)

El ejercicio de obtener las (ELS) y las (ELD), nos lleva a la necesidad de refinar mejor nuestros procedimientos, utilizando para esto el concepto BASE DE DATOS LOGICA, la cual es posible obtener a partir de las jerarquías definidas en las ELS y ELD, utilizando además como complemento, el método de normalización desarrollado por E. F. CODD (GANE Y SARSON, 1977) el cual envía las redundancias de datos y los errores de actualización, evidentes en un archivo secuencial, como en el caso mostrado al definir la ELD en el dato "NOMBRE DEL VENDEDOR" y si las sucursales son múltiples sería necesario también recurrir al conocimiento de su descripción.

La base de datos lógica, es la organización de las estructuras lógicas de datos, sin redundancia, identificadas en un medio ambiente, relacionadas entre si mediante llaves.

Una llave es la cantidad mínima de datos que identifica a cada registro definido en las (ELD). El proceso secuencial que se siguen para lograr la (BDL) es el siguiente:

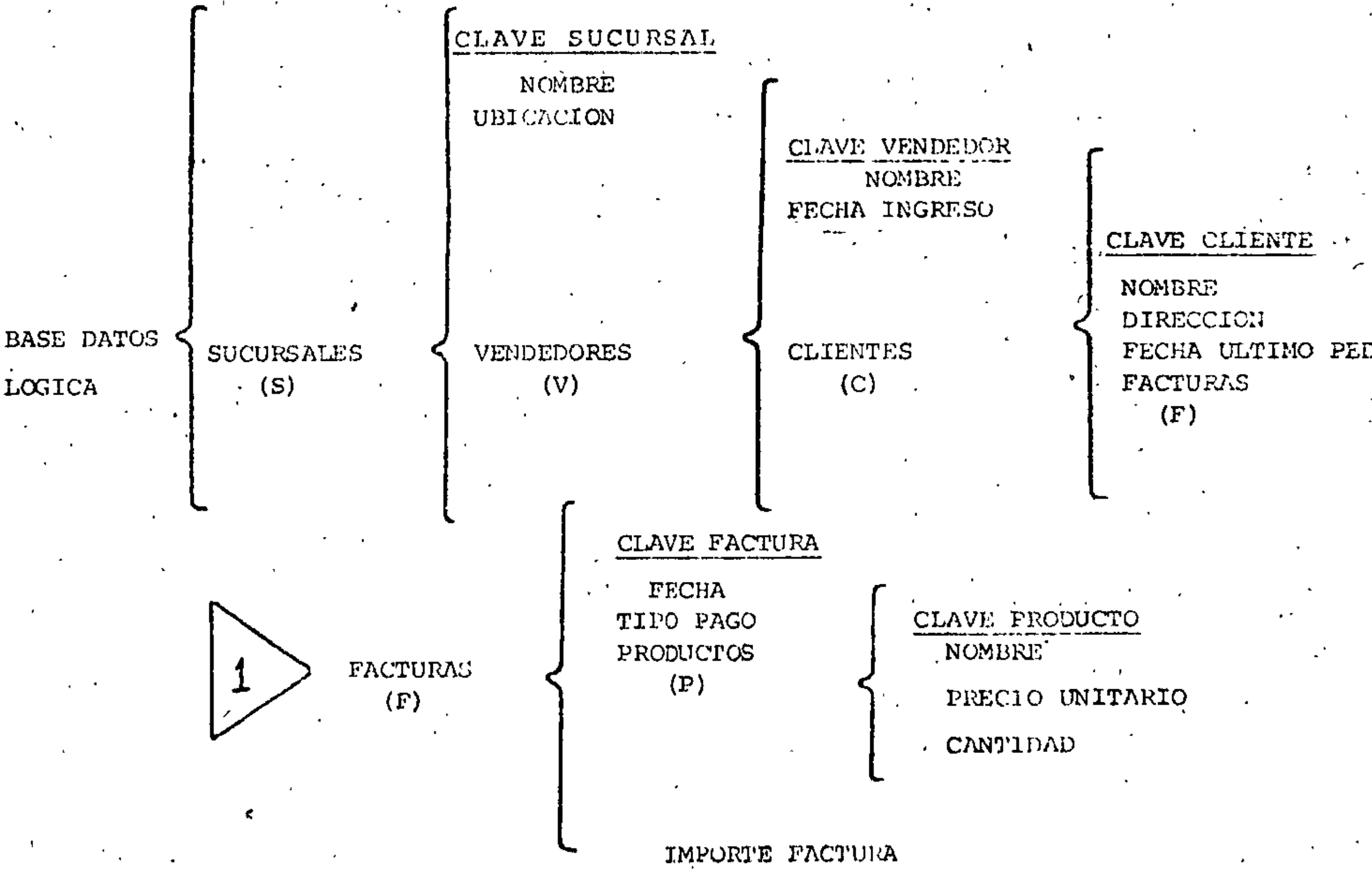
1) Cada conjunto repetitivo deberá formar una estructura lógica de datos.

2) Cada conjunto de datos NO-LLAVE que dependa de una parte de la llave deberá formar una nueva estructura lógica de datos.

3) Identificadas las estructuras lógicas de datos, con los dos refinamientos anteriores, ningún dato NO-LLAVE podrá derivarse de otros datos NO-LLAVE.

Respectivamente a los pasos anteriores se les denomina, primera, segunda y tercera formas normales.

Como extensión a nuestro problema anterior supongamos una base de datos lógica, jerarquicamente estructurada.



El dato CLAVE SUCURSAL, con el subrayado representa una llave, si no hemos procedido a hacer ningún refinamiento, diríamos que solo se trata de LLAVE CANDIDATA, a los -- otros datos se les denomina NO-LLAVE. La llave puede ser simple si un solo elemento sirve para identificar datos - NO-LLAVE, es COMPUESTA si requiere de mas de un elemento para identificar a cada dato NO-LLAVE, los datos llave se subrayan y los NO LLAVE se dejan igual.

El primer refinamiento daría una ordenación como la siguiente:

- BASE DATOS LOGICA
  - (ELD) SUCURSALES
    - SUCURSAL (S)
      - CLAVE SUCURSAL
      - NOMBRE
      - DIRECCION
  - (ELD) VENDEDORES
    - VENDEDOR (V)
      - CLAVE SUCURSAL
      - CLAVE VENDEDOR
      - NOMBRE
      - FECHA INGRESO CIA.
  - (ELD) CLIENTES
    - CLIENTE (C)
      - CLAVE SUCURSAL
      - CLAVE VENDEDOR
      - CLAVE CLIENTE
      - NOMBRE
      - DIRECCION
      - FECHA ULTIMO PEDIDO
  - (ELD) FACTURAS
    - FACTURA (F)
      - CLAVE SUCURSAL
      - CLAVE VENDEDOR
      - CLAVE CLIENTE
      - NO. FACTURA
      - IMPORTE
  - (ELD) PRODUCTOS
    - PRODUCTOS VENDIDOS (PV)
      - CLAVE SUCURSAL
      - CLAVE VENDEDOR
      - CLAVE CLIENTE
      - CLAVE PRODUCTO
      - NOMBRE PRODUCTO
      - PRECIO UNITARIO
      - CANTIDAD VENDIDA

Existen en los (ELD) definidos datos que solo dependen de una parte de la llave o que dependen de un dato NO-LLAVE, describimos las observaciones hechas a la estructura anterior.

SUCURSAL
(S)

- CLAVE SUCURSAL
- NOMBRE
- DIRECCION

VENDEDOR
(V)

- CLAVE VENDEDOR
- NOMBRE
- FECHA INGRESO COMPAÑIA
- CLAVE SUCURSAL ASIGNADO

CLIENTE
(C)

- CLAVE CLIENTE
- NOMBRE
- DIRECCION
- STATUS

CARTERA CLIENTES
POR SUCURSAL
(CC)

- CLAVE SUCURSAL
- CLAVE CLIENTE
- FECHA ULTIMO PEDIDO

FACTURAS
(F)

- CLAVE SUCURSAL
- No. FACTURA
- FECHA
- TIPO PAGO
- IMPORTE
- CLAVE VENDEDOR
- CLAVE CLIENTE

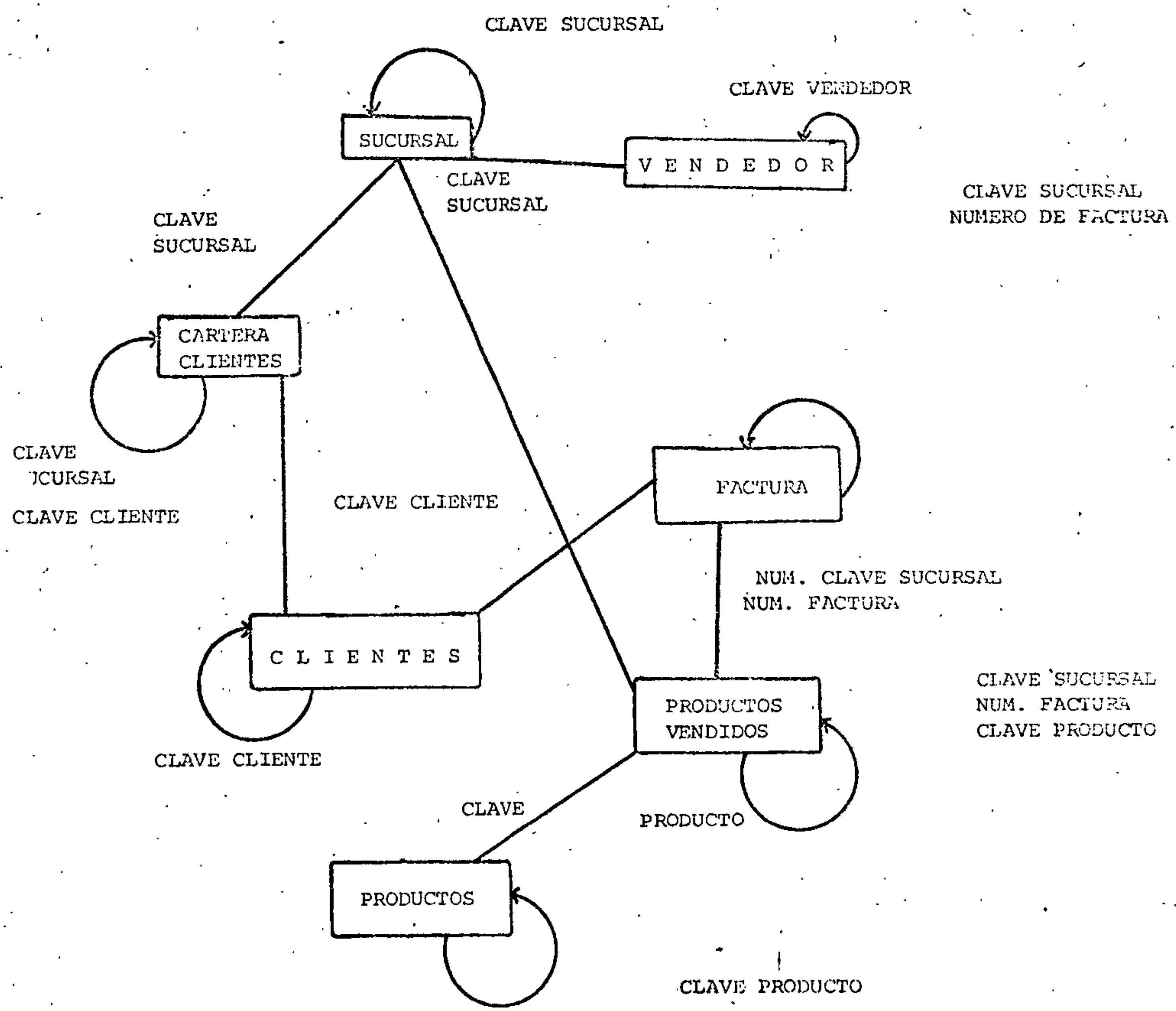
PRODUCTOS
(P)

- CLAVE PRODUCTO
- NOMBRE PRODUCTO
- PRECIO UNITARIO

PRODUCTOS
VENDIDOS
(PV)

- CLAVE SUCURSAL
- CLAVE PRODUCTO
- FACTURA
- CANTIDAD VENDIDA

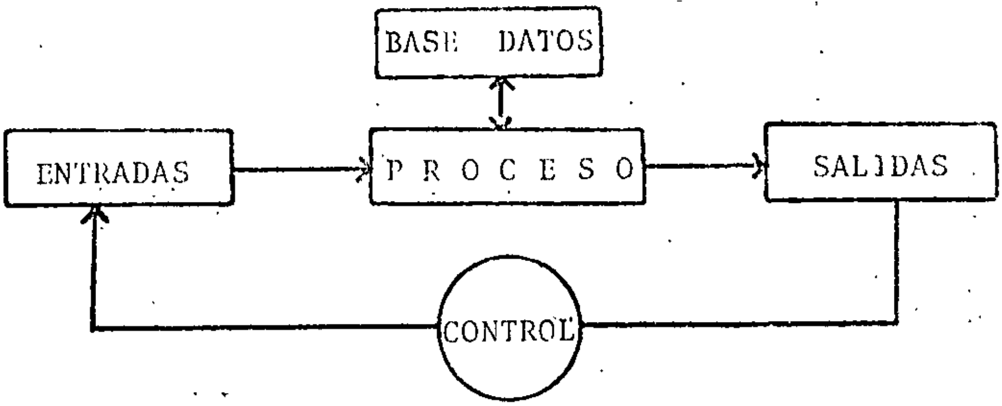
Un esquema de relaciones y llaves ayuda a tener una mejor idea de como recuperar datos, ya sea en la misma estructura de datos lógica o mediante asociaciones, de llaves y no-llaves.



Algunas de las funciones, aplicables son:

CODIFICAR, CAPTURAR, VALIDAR, REGISTRAR, ACTUALIZAR, PROCESAR, CARGAR, RESPALDAR, RECUPERAR, CALCULAR, PROYECTAR, PRESENTAR, GRAFICAR, RASTREAR, ENVIAR, ETC.

Con respecto al flujo de datos esto se relaciona al modelo básico de, entradas, salidas, base de datos y control, asociados a un mecanismo de reconocimiento y transformación, denominado proceso o caja negra, que en nuestro caso son las funciones.



El modelo presentado reúne una solución lógica que debemos desarrollar progresivamente, definiendo las estructuras lógicas de salida, las estructuras lógicas de datos, y la base de datos lógica; ahora solo falta definir, el proceso y los mecanismos de control que son precisamente las funciones que transforman datos en infrormación, y -

DIAGRAMA DE FUNCIONES Y FLUJO DE DATOS

Los beneficios de poner en un papel y lápiz una futura estructura, tiene una clara economía de recursos, ya que primero se piensa en que se van a usar y después se obtienen y gastan, -- consideramos que el predominio de lo lógico sobre lo físico, debe mantenerce como una disciplina profesional, que como resultado proporcionará una alta calidad a nuestro desarrollo.

Una herramienta mas, derivada del uso de los diagramas WARNIER/ORR, recibe el nombre de diagrama de funciones y flujo de datos, en este instrumento se integra un modelo de solución de un alto grado de refinamiento, donde se describen las funciones manuales y automáticas dirigidas al proceso de un sistema informático; relacionando cada función, con su correspondiente flujo de datos.

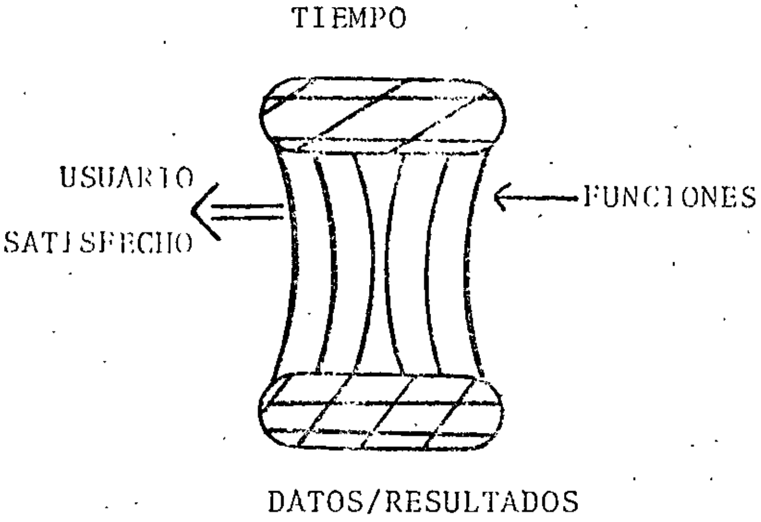
En informática una función es; o una acción o un grupo de acciones, particularmente identificadas con los mecanismos tradicionales del proceso de datos o bien con lo mas sofisticado de la automática y el control, es decir, se puede adaptar a las necesidades mas simples o complejas.

por otra parte, garantizan que el sistema mantenga su equilibrio aprovechando al máximo la sinergía (cooperación) de las partes, para lograr que el todo sea mejor que la suma - de partes.

Un concepto de diseño importante indica que una buena solución es aquella que logra, producir el máximo de satisfactores con el mínimo de sacrificios, en el contexto informático buscamos obtener las salidas corréctas, con los datos correctos en tiempos correctos.

Consideramos que de esta forma, logramos el máximo de satisfactores, con el mínimo de recursos; en datos, almacenados, recuperados, examinados en los tiempos oportunos. Para nosotros el tiempo juega el papel mas importante en el diseño de funciones ya que los resultados para el usuario deben -- ser oportunos, no se pueden posponer informes y pagos, ya que esto provocaría graves sanciones para la organización y por esta causa la imágen de la informática se vería afectada por una falta de confianza; de hecho, muchas organizaciones sienten este fenómeno y fundamentalmente se debe a que la definición de requerimientos y el diseño hicieron caso omiso del tiempo.

Con respecto a los datos y las funciones el tiempo juega dimensión que tratamos de mostrar mediante una malla entrelazada donde las funciones aplicadas en los datos, proporcionan los resultados en el tiempo correcto.



Además de considerar el tiempo como esencial para la solución de problemas informáticos, debemos decir que el diagrama de funciones y flujo de datos apoya enormemente el trabajo de documentación y comunicación con el usuario.

En forma gráfica muestra cual es la descomposición del --sistema y cada parte que resultados entrega; la documentación obtenida, se va mejorando adaptándole progresivamente los archivos y programas que se desarrollan, además señala los procedimientos que se deben ejecutar en relación al tiempo, este instrumento por la presentación que tiene, puede ser cuestionado por el usuario dándole la posibilidad de participar en las revisiones que garanticen que en verdad es el producto que desea; para nosotros, muchos de los problemas y errores del sistema se deben a la falta de participación del usuario en la solución y esto en muchas ocasiones se debe a que el instrumento donde se presenta - la solución utiliza un lenguaje díficil de transmitir; de hecho, cuando él usuario no comprende algunas palabras y - simbolos técnicos, que para el informático son triviales, la reacción inmediata es una falta de interés, debido a que siente un acto de superioridad o que le están engañando.

El éxito de WARNIER/ORR radica en comunicar claramente los planes de solución que tiene el informático para el usuario, en un lenguaje sencillo y accesible que facilite la - participación y la comprensión.

Un procedimiento queda claro cuando objetivamente se muestran las respuestas a las preguntas siguientes:

- Qué se quiere hacer ?

  La respuesta generalmente se asocia a la identificación de los objetivos y las acciones ordenadas.

- Porqué se quiere hacer ?

  La respuesta implica una asociación con los productos que se desea obtener.

- Como se quiere hacer ?

  La respuesta está asociada a la forma en la que el sistema resuelve el problema, ya sea con medios automatizados o con mecanismos manuales.

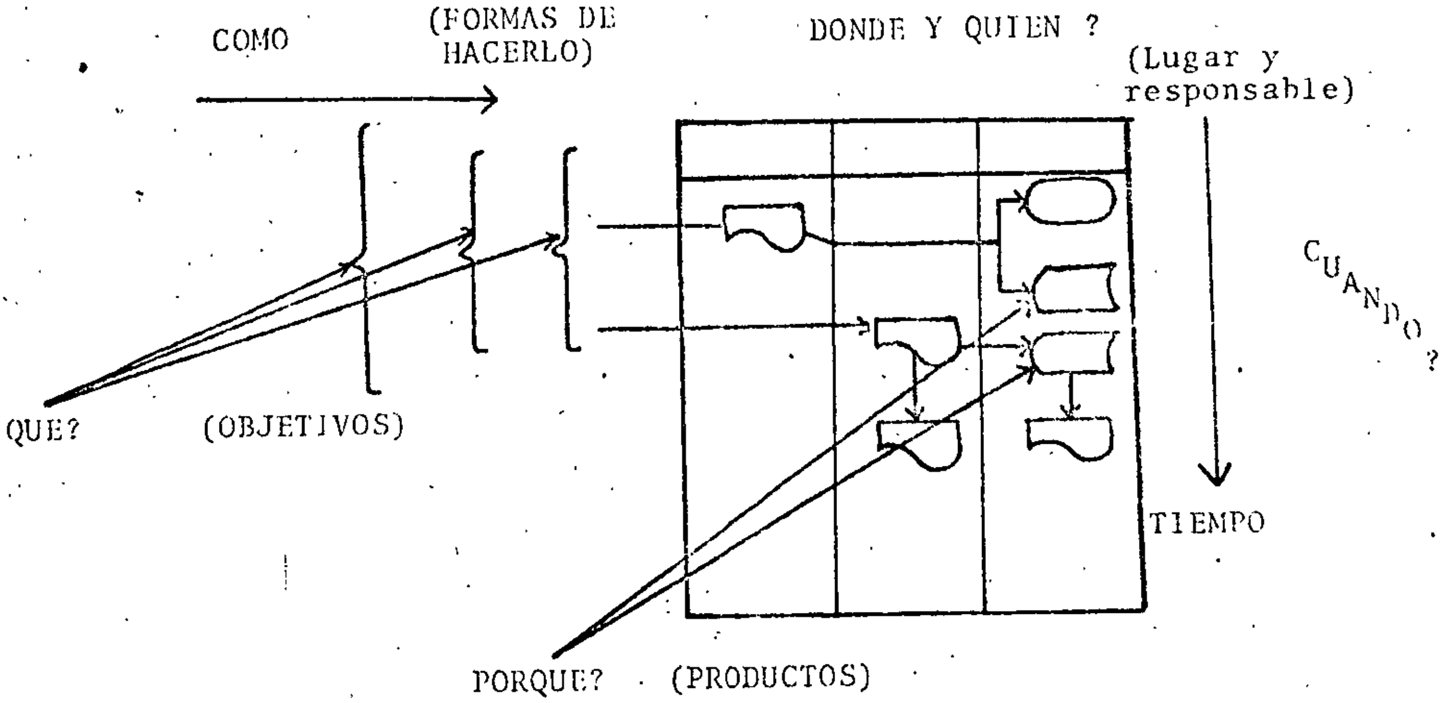
- Quién y donde se hace ?

  Indica los responsables y el lugar, en el que la función se lleva a cabo.

- Cuando se debe hacer ?

  Indica el tiempo en el que se ejecuta la función, generalmente se ve relacionada a las jerarquías y a las secuencias.

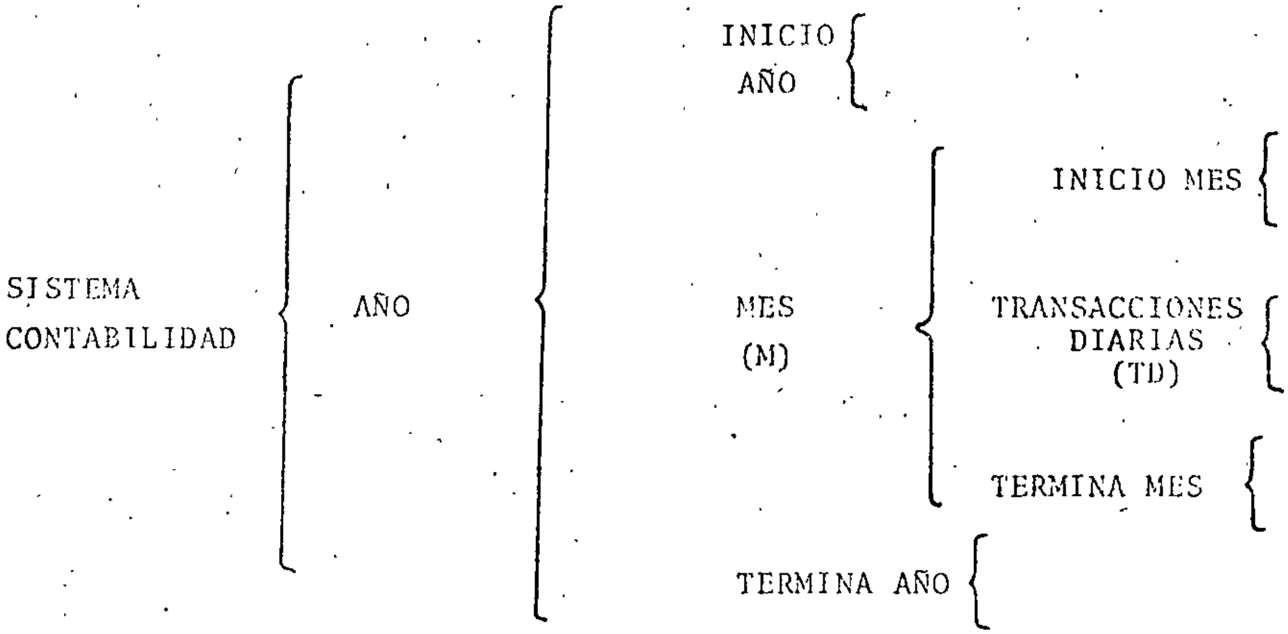
Graficamente en un diagrama de funciones y flujo de datos, las preguntas se asocian de la siguiente forma:



Las funciones se ordenan mediante jerarquías asociadas al tiempo, para lo cual se descomponen en tres conjuntos; 1) que corresponde a procesos de inicio, 2) una nueva repetición u otra secuencia central que indica un nuevo proceso y 3) una parte final que referida a los procesos de - terminación

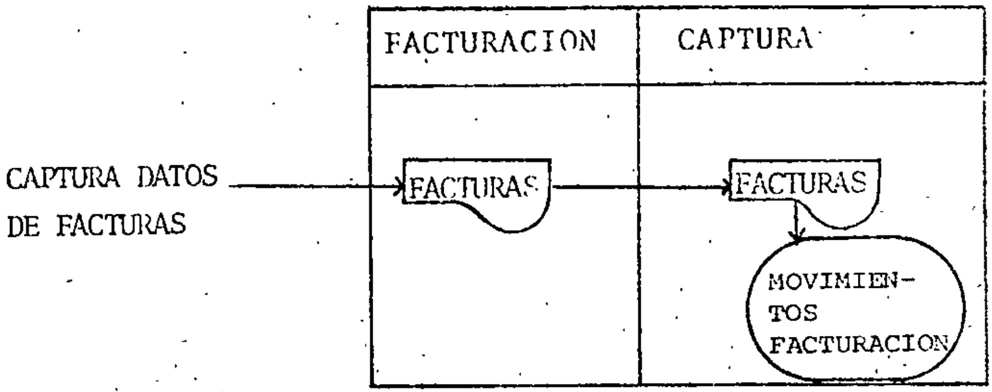
Las funciones se realizan mediante construcciones básicas, jerarquías y/o secuencias y/o repeticiones y/o alternaciones.

Por ejemplo un sistema de contabilidad típico, tiene un proceso anual integrado atraves de las operaciones mensuales, las cuales a su vez se integran del proceso de transacciones, los conjuntos representativos están jerarquizados en la siguiente estructura:



Cada año, se descompone en una étapa de inicio, los meses y el fin, igualmente sucede con los meses y con las transacciones diarias, al proceder de una manera, tenemos como resultado una ordenación clara y precisa de, la formación del sistema.

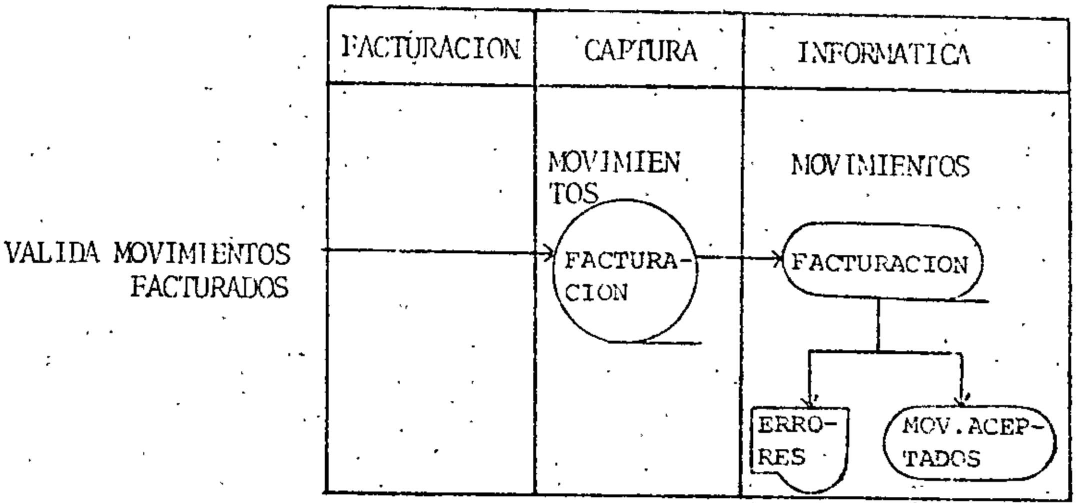
De la misma manera, cada conjunto generado puede razonarce descomponiendolo en inicio, parte media y fin, a este proceso le denominamos análisis y a la generación de un nuevo conjunto le llamamos síntesis, toda función tiene como consecuencia una interacción con los datos, estos se presentan mediante un flujo ya sea de entrada y/o salida, sucesivamente una salida puede ser la entrada a una nueva función.



La función "CAPTURA DATOS DE FACTURAS" se realiza tomando como entrada los documentos fuente del departamento de facturación, quien los pasa al departamento de captura y en -

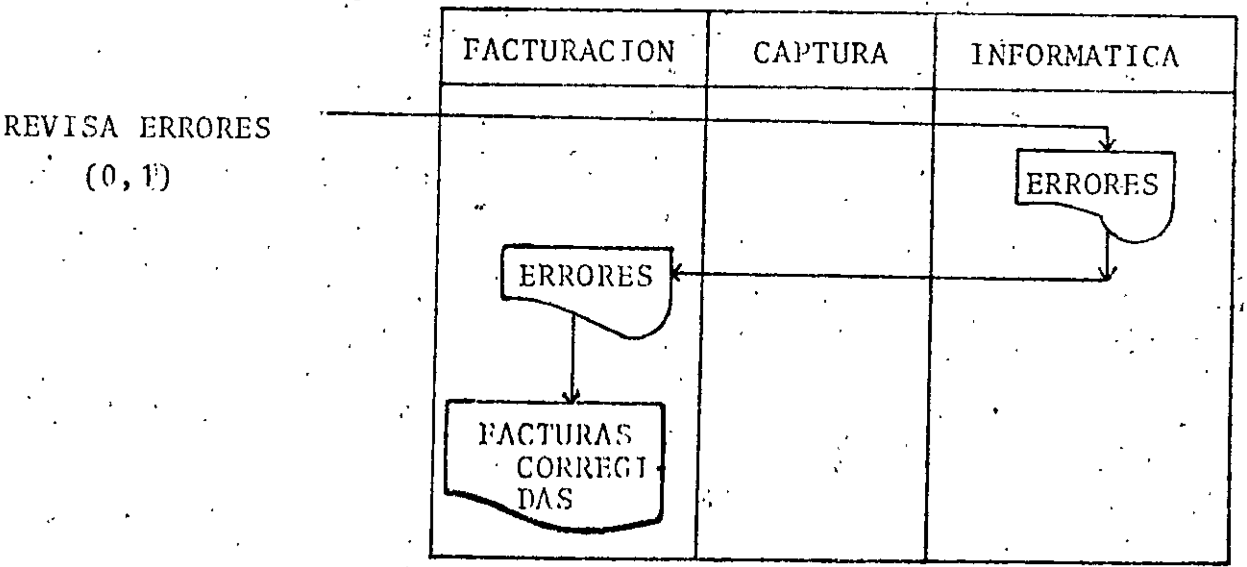
ese lugar precisamente se obtiene una cinta de "MOVIMIENTOS DE FACTURACION".

Una segunda función muestra un proceso de validación.

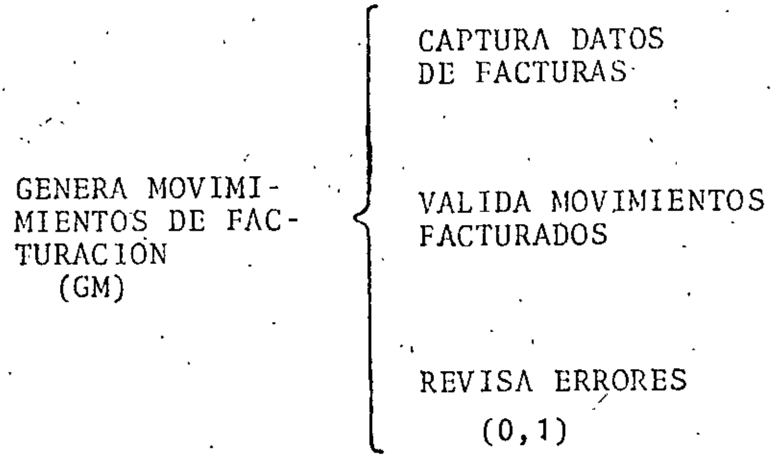


La función "VALIDA MOVIMIENTOS FACTURADOS", se realiza mediante los -- MOVIMIENTOS DE FACTURACION generados en el departamento de captura, y los cuales en el departamento de informática, sirven para generar un - "REPORTE DE ERRORES" y una cinta de "MOVIMIENTOS ACEPTADOS".

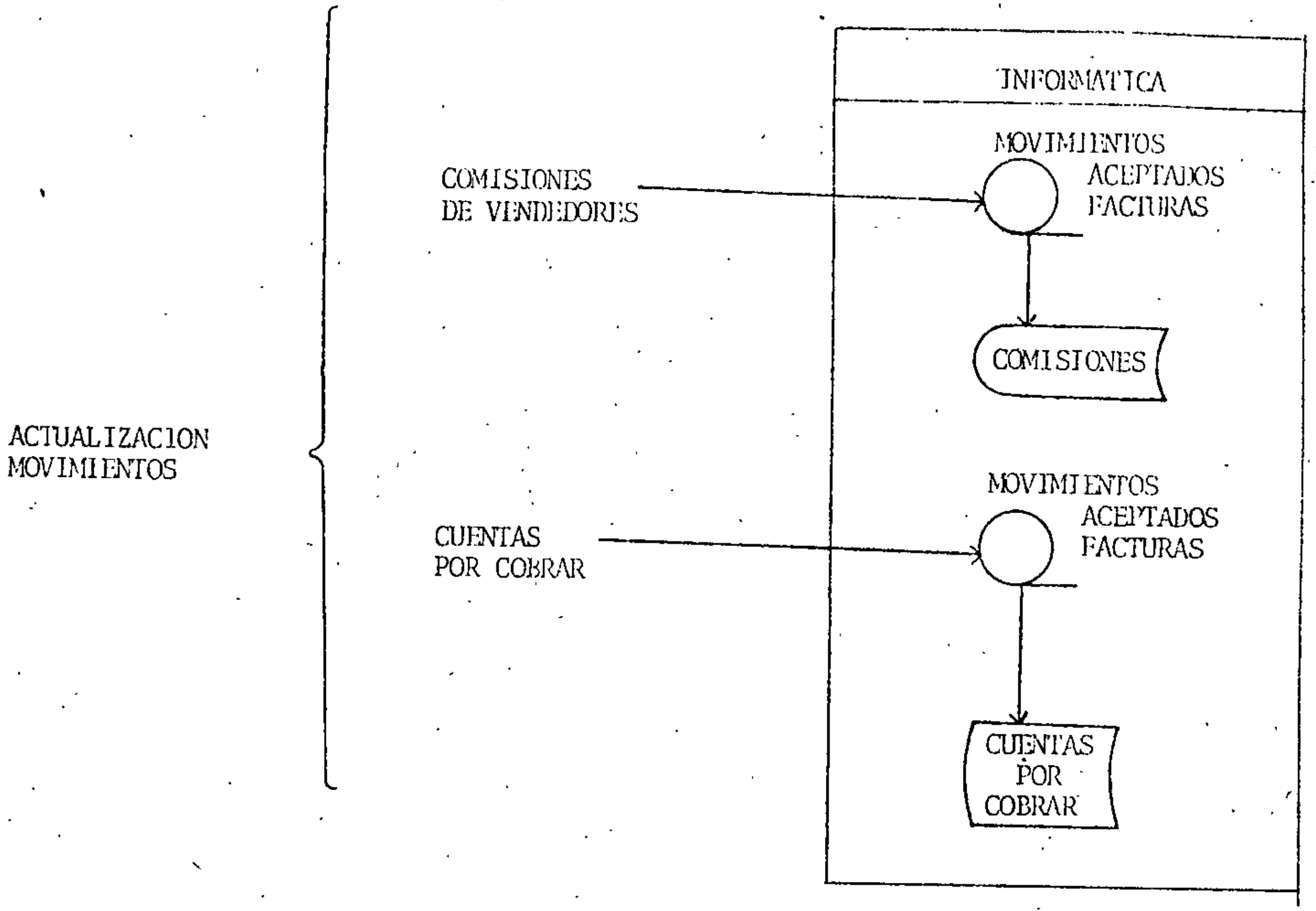
La función siguiente es "REVISAR ERRORES":



La función "REVISA ERRORES" puede o no realizarce, si no se realiza esto se debe a que no existen errores, la observación de los errores generados en informática, se pasan al departamento de facturación y ahí se generan "FACTURAS CORREGIDAS", las funciones anteriores las reunimos en un conjunto denominado "GENERA MOVIMIENTOS DE FACTURACION".



En el anterior conjunto observamos que existe una repetición, (GM) veces, la secuencia lógica es la actualización de movimientos, primero esto puede hacerce para pagar comisiones a los vendedores y segundo para abrir cuentas -- por cobrar a clientes, la forma de representar esto sería:



Secuencialmente, vamos enlazando funciones formando nuevos conjuntos, mediante un proceso de análisis y síntesis.

De los ejemplos anteriores notamos que las funciones las deducimos de arriba hacia abajo (Sistema Contabilidad), o también de abajo hacia arriba (funciones de facturación), finalmente ambos métodos llegan al mismo resultado; el objetivo es integrar las funciones y reconocer como se va dando el flujo de datos, debemos notar también que al final del proceso observamos como las partes constituyen el todo y que solo de esa manera es posible desarrollar el sistema correctamente, la utilización mas importante del diagrama de funciones y flujo de datos está dirigida a la comunicación con los usuarios y las áreas involucradas para la integración del sistema, además adquiere un valor insuperable conforme sirve para capacitar al usuario e inclusive a los nuevos miembros que se van incorporando al desarrollo, en la fase de producción evita errores y olvidos no intencionados.

Pudiéramos decir que es un instrumento apropiado para definir procedimientos del manejo de información, ya que relaciona correctamente las funciones y los datos, y no solo eso, además muestra las secuencias y decisiones tomadas, en los tiempos correctos.

ESTRUCTURACION LOGICA DE PROCESOS (ELP)

La consecuencia de estructurar lógicamente las salidas, los - datos, las bases de datos y las funciones; son los procesos - efectuados, es decir las acciones efectivas que permiten realizar la transformación de datos de entrada en datos de salida, las acciones se ordenan en las estructuras lógicas de proceso, éstas se forman mediante un verbo imperativo y un complemento directo, por ejemplo:

Suma	total-hombres-municipio a total-hombres-estado
Mueve	ceros a total-hombres-municipio
Escribe	"total de ventas"
Lee	primer registro de movimientos de almacén
Examina	punto de reorden
Analiza	transacciones

De esta manera expresamos, instrucciones que transforman datos, y como resultado logramos el diseño lógico de un programa.

El nombre de estructuración lógica se debe a que el proceso (conjunto de acciones) se hace mediante construcciones básicas, mapeando, los requerimientos de entrada a salida, considerando las reglas de cálculo y las constantes de salida; como las funciones, los procesos comienzan, mediante una descomposición de las jerarquías en; inicio, parte media y fin (análisis), cada conjunto formado (síntesis), se le aplica el mismo tratamiento hasta llegar al detalle que facilite el mapeo físico a las instrucciones de algún lenguaje de programación.

Recordemos que conociendo la estructura de datos, la estructura del proceso es su derivación lógica, sea por -- ejemplo que simplemente queremos listar un archivo:

ESTRUCTURA LOGICA DE DATOS

- REGISTRO ENTRADA
  - REGISTRO (R)
    - DATO 1
    - DATO 2
    - DATO 3

ESTRUCTURA LOGICA DE PROCESO

- PROGRAMA
  - INICIA PROGRAMA
    - ABRE ARCHIVOS
    - MUEVE CEROS A CIFRAS CONTROL
    - ESCRIBE "LISTADO"
    - LEE PRIMER REGISTRO
  - PROCESA REGISTRO (R)
    - MUEVE DATOS ENTRADA A DATOS SALIDA
    - IMPRIME REGISTRO DE SALIDA
    - LEE SIGUIENTE REGISTRO
  - TERMINA PROGRAMA
    - IMPRIME CIFRAS CONTROL
    - CIERRA ARCHIVOS

ESTRUCTURA LOGICA DE SALIDA

- ARCHIVO SALIDA
  - ENCABEZADO
    - "LISTADO"
  - REGISTRO (R)
    - DATO 1
    - DATO 2
    - DATO 3
  - CIFRAS CONTROL
    - LEIDOS
    - IMPRESOS

La forma clásica de representar un programa mediante diagrama de flujo, en la forma sugerida por Warnier es:

DIAGRAMA WARNIER

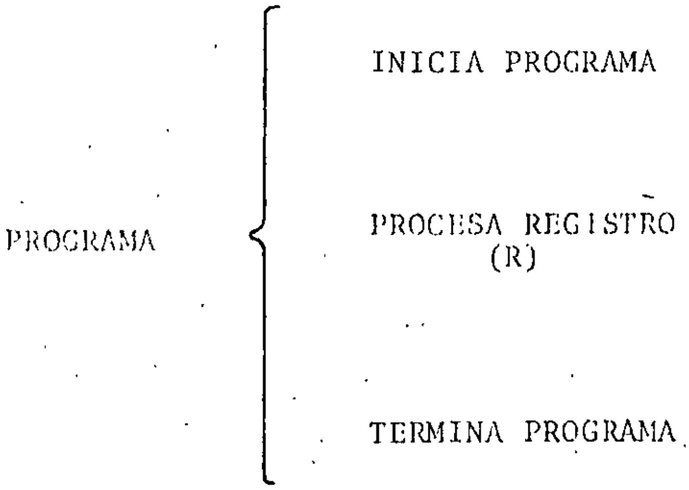
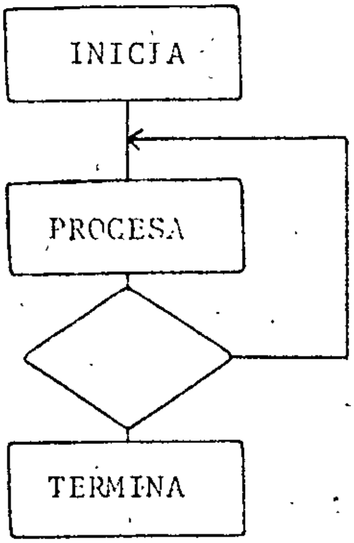


DIAGRAMA DE FLUJO WARNIER



Un mapeo físico directo a "COBOL" (lenguaje de programación reconocido para aplicaciones en cualquier clase de negocios), sería en la parte de "Procedure División".

```
Procedure division

Genera-listado

        Perform  inicia-programa
        Perform  procesa-registro
                 until
                 (B-fin-registro egual verdadero)
        Perform
        stop run
```

Una aclaración importante; cuando diseñamos la estructura lógica de proceso debemos tener presente lo "lógico antes de lo físico"; es decir, primero tenemos que estar seguros que lógicamente es lo que queremos; problemas físicos como el salto de hoja, uso de formateadores de pantallas, editores, compiladores excelentes, lenguajes "mas científicos"; no deben restringir lo lógico, creemos que este tipo de detalles, son la causa de muchos errores, y de hecho; hasta hoy, le hemos dado importancia al diseño, debido a que, libros, maestros y escuelas, eran propagandistas del CODIGE, de los lenguajes, de que tal máquina era lo máximo en arquitectura, y realmente vemos que antes de cualquier solución debemos plantear el problema, porque sin duda; un problema no planteado es un problema no resuelto.

Para continuar con nuestra exposición, de estructuras lógicas de proceso y comprender más a detalle como se utiliza la herramienta, definiremos los pasos para lograr el (ELP).

1) Definir la estructura lógica de salida.

2) Definir la estructura lógica de datos.

   2.1) Listar los datos señalando constantes, cálculos y datos primarios.

   2.2) Obtener en base a los datos primarios, la estructura lógica de datos, o ENTRADA IDEAL.

3) Identificar las jerarquías que deben ejecutarse de la entrada ideal.

4) Mapear las jerarquías a una estructura lógica de proceso, señalando que se trata de un proceso.

5) Agregar a cada conjunto jerárquico dos conjuntos, uno de inicio y otro de terminación.

6) Si la entrada tiene estructuras alternativas, éstas se descomponen en una estructura lógica alternativa formando un conjunto de inicio, una parte donde se examina, analiza, compara, selecciona; la alternativa y un conjunto de terminación, generalmente en algunos casos los conjuntos de inicio y terminación pueden estar vacíos, si esto sucede se pueden omitir o colocar la frase "NULO".

Cada conjunto se va llenando de acuerdo a los siguientes criterios:

1) En los conjuntos de inicio.

1.1) En el primero de izquierda a derecha se deben abrir archivos.

1.2) Se deben iniciar a cero los acumuladores.

1.3) Se deben imprimir los encabezados y datos referidos en el nivel de salida.

1.4) Se deben guardar las claves de control para efectuar el cambio.

1.5) En el primer conjunto de izquierda a derecha se debe hacer una lectura inicial al primer registro ideal.

1.6) Se deben poner en falso las banderas de control para efectuar el cambio.

2) En los conjuntos de terminación.

2.1) Se deben imprimir los párrafos de totales correspondientes al nivel de salida.

2.2) Se deben imprimir los datos sumarizados en ese nivel.

2.3) Si no es el primero de izquierda a derecha, se deben sumar los sumadores al nivel izquierdo - anterior.

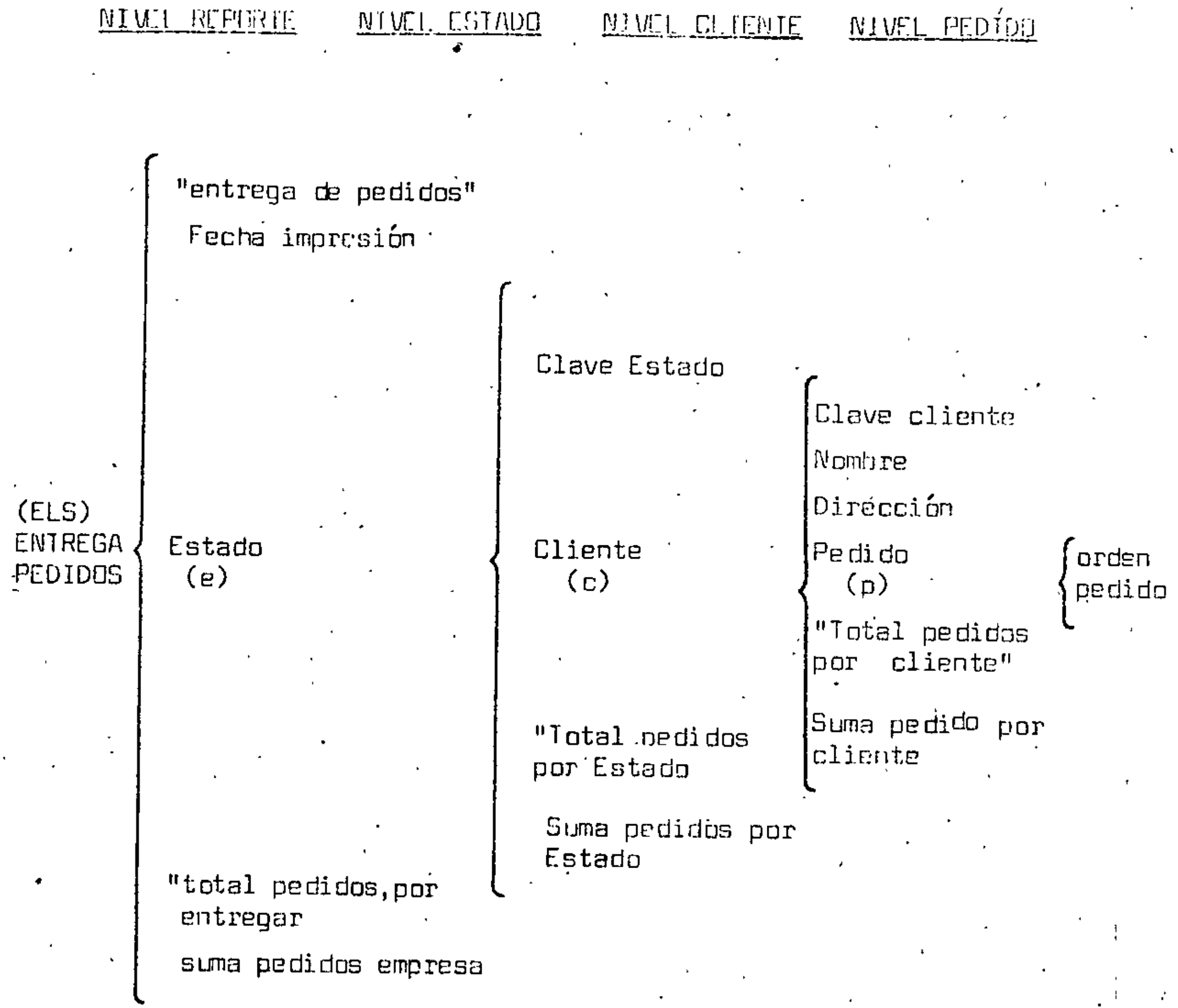
2.4) Si es el primero de izquierda a derecha se deben enviar las cifras de control y cerrar archivos.

3.- El último conjunto de la parte media, debe considerar después de ejecutar las operaciones correspondientes a ese nivel, lèer el siguiente registro.

Para ilustrar concretamente como desarrollar una estructura lógica de proceso, supongamos que se desean entregar pedidos a los diferentes estados de la República mediante una lista que muestre la clave de clientes, nombre del cliente, dirección por Estado, así como la orden del pedido o varios pedidos si es el caso, contando cuantos pedidos existen por cliente, por Estado y la empresa.

Físicamente un esqueleto que muestra esto sería:

```
|Fecha impresión|     "entrega de pedidos"
|Estado            |
|Clave cliente| |Nombre cliente| |Direccion |  |   Pedido   |
                                                 |____________|
                              Total pedidos      |____________|
|Clave cliente| |______________| |__________|   |____________|
                              Total pedido       |____________|
                Total pedidos por Estado         |____________|
|____________|
                |______________| |__________|   |____________|
                              Total pedido       |____________|
                Total pedidos por  Estado        |____________|
                Total pedidos por entregar       |____________|
```


NIVEL REPORTE NIVEL ESTADO NIVEL CLIENTE NIVEL PEDIDO

(ELS)
ENTREGA
PEDIDOS

"entrega de pedidos"
Fecha impresión

Estado
(e)

Clave Estado

Cliente
(c)

Clave cliente
Nombre
Dirección
Pedido
(p)

orden
pedido

"Total pedidos
por cliente"

Suma pedido por
cliente

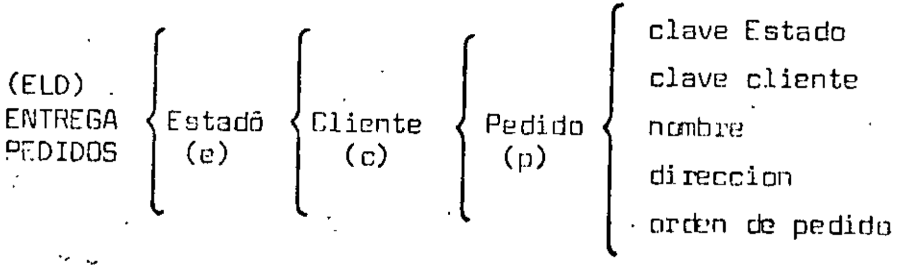
"Total pedidos
por Estado

Suma pedidos por
Estado

"total pedidos,por
entregar

suma pedidos empresa

MAPEAMOS A LA ESTRUCTURA LOGICA DE DATOS


(ELD)
ENTREGA
PEDIDOS

Estadô
(e)

Cliente
(c)

Pedido
(p)

clave Estado
clave cliente
nombre
direccion
orden de pedido

Damos por entendido que las constantes y los datos calculados quedaron identificados y solamente se referencían en la ELD - los datos primarios.

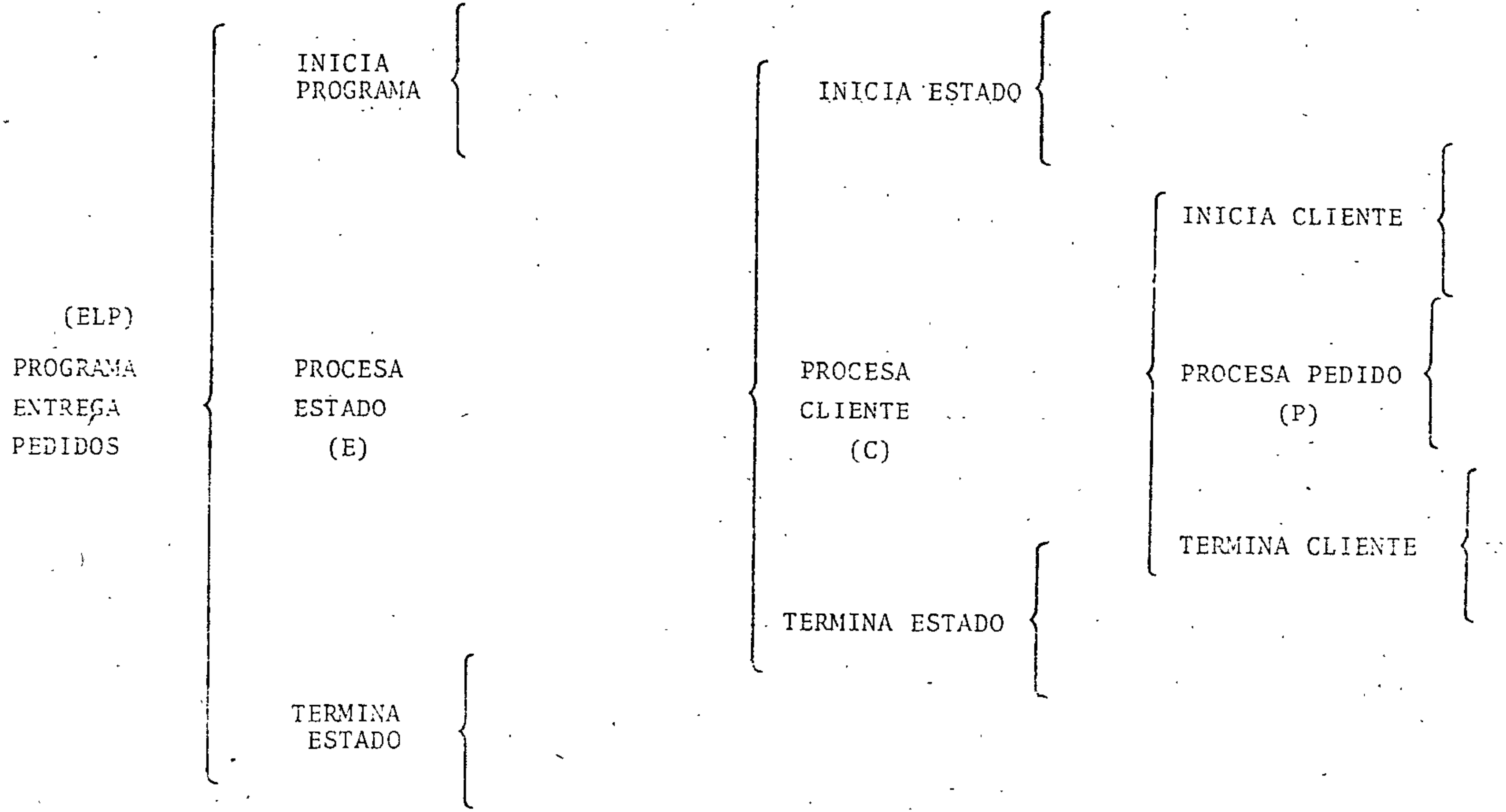
Denominamos a la siguiente estructura "ENTRADA IDEAL"

$$\begin{matrix}\text{PEDIDO}\\ \text{(1)}\end{matrix} \begin{cases} \text{FECHA} \\ \text{CLAVE ESTADO} \\ \text{CLAVE CLIENTE} \\ \text{NOMBRE} \\ \text{DIRECCION} \\ \text{ORDEN PEDIDO} \end{cases}$$

La estructura lógica proceso comienza con el mapeo de las jerarquías de entrada.

$$\begin{matrix}\text{(ELP)}\\ \text{PROGRAMA}\\ \text{ENTREGA}\\ \text{PEDIDOS}\end{matrix} \left\{ \begin{matrix}\text{PROCESA ESTADO}\\ \text{(E)}\end{matrix} \left\{ \begin{matrix}\text{PROCESO}\\ \text{CLIENTE}\\ \text{(C)}\end{matrix} \left\{ \begin{matrix}\text{PROCESA}\\ \text{PEDIDO}\\ \text{(P)}\end{matrix} \right.\right.\right.$$

A cada conjunto le agregamos las acciones de inicio y terminación.



- (ELP) PROGRAMA ENTREGA PEDIDOS
  - INICIA PROGRAMA
  - PROCESA ESTADO (E)
    - INICIA ESTADO
    - PROCESA CLIENTE (C)
      - INICIA CLIENTE
      - PROCESA PEDIDO (P)
      - TERMINA CLIENTE
    - TERMINA ESTADO
  - TERMINA ESTADO

Llenamos los conjuntos de acuerdo a las consideraciones dadas.

Como observamos el control de la repetición en cada jerarquía, la hemos controlado mediante la proposición HASTA, lógicamente vemos como a cada nivel le corresponde el más alto control; es decir, que se acaben los registros que contiene la (ELD) en el nivel programa, en el nivel estado, el control se hace agregando al control del nivel izquierdo anterior un control mas, precisamente para ese nivel; en este caso se da mediante la diferencia entre la clave estado leída y la w-clave-estado guardada, razonando del mismo modo, usamos en el nivel cliente los controles del nivel izquierdo anterior y el control del nuevo nivel, se da mediante la diferencia entre clave cliente leída y W-clave-cliente guardada.

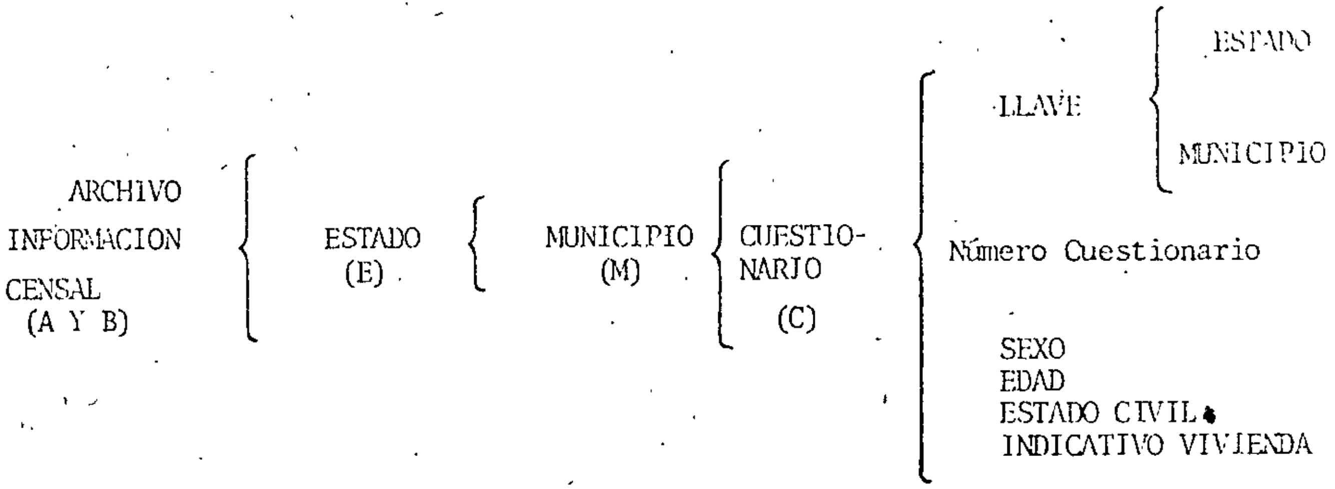
También observamos que en el primer conjunto izquierdo y el último conjunto de en medio al final de estos aparece el conjunto "EXAMINA FIN REGISTROS", el cual sirve para una terminación no esperada del archivo, en el primer caso revisa que no exista ningún registro, y en el último reconoce un fin de archivo tradicional.

Una revisión lógica al instrumento obtenido nos lleva a la conclusión de que la solución es correcta, esto nos lleva a decir que el método Warnier es algo más que hacer un programa; simplifica el trabajo, ayuda á no tener errores, a transmitir mejor nuestras ideas, el trabajo de pruebas y compilaciones es mínimo y la codificación no se vuelve ambigua, obviamente no todos los problemas se resuelven con Warnier; pero razonando un poco vemos que es un cambio para hacer del desarrollo de sistemas informáticos una ciencia.

Tampoco vamos a decir que todos los algoritmos existentes se resuelven mediante el método Warnier; sin embargo, vemos una cosa real, sí podemos expresar cualquier algoritmo con los diagramas Warnier; y esto, para nuestra indiosincracia, necesidades, idioma y disciplina, es valioso, tenemos hoy un método y un conjunto de simbolos que pueden hacer mas fácil el desarrollo, administración y documentación de sistemas informáticos, por lo tanto provoca que la programación pase de una práctica privada a una práctica - pública; en este terreno, se acabó el elitismo y la soberanía del programador egoísta y en su lugar emerge el humilde programador que desea que su conocimiento sea respetable, honorable y de alta calidad profesional.

Mostraremos ahora una solución dirigida a mezclar dos archivos lógicos de entrada previamente clasificados y cuyo objetivo es obtener un tercero que contenga la información de los otros dos, mediante una llave, que contiene los datos estado y municipo, además contiene los datos, número cuestionario, sexo, edad, estado civil, indicativo de vivienda.

La estructura lógica de datos de los dos archivos de entrada es:

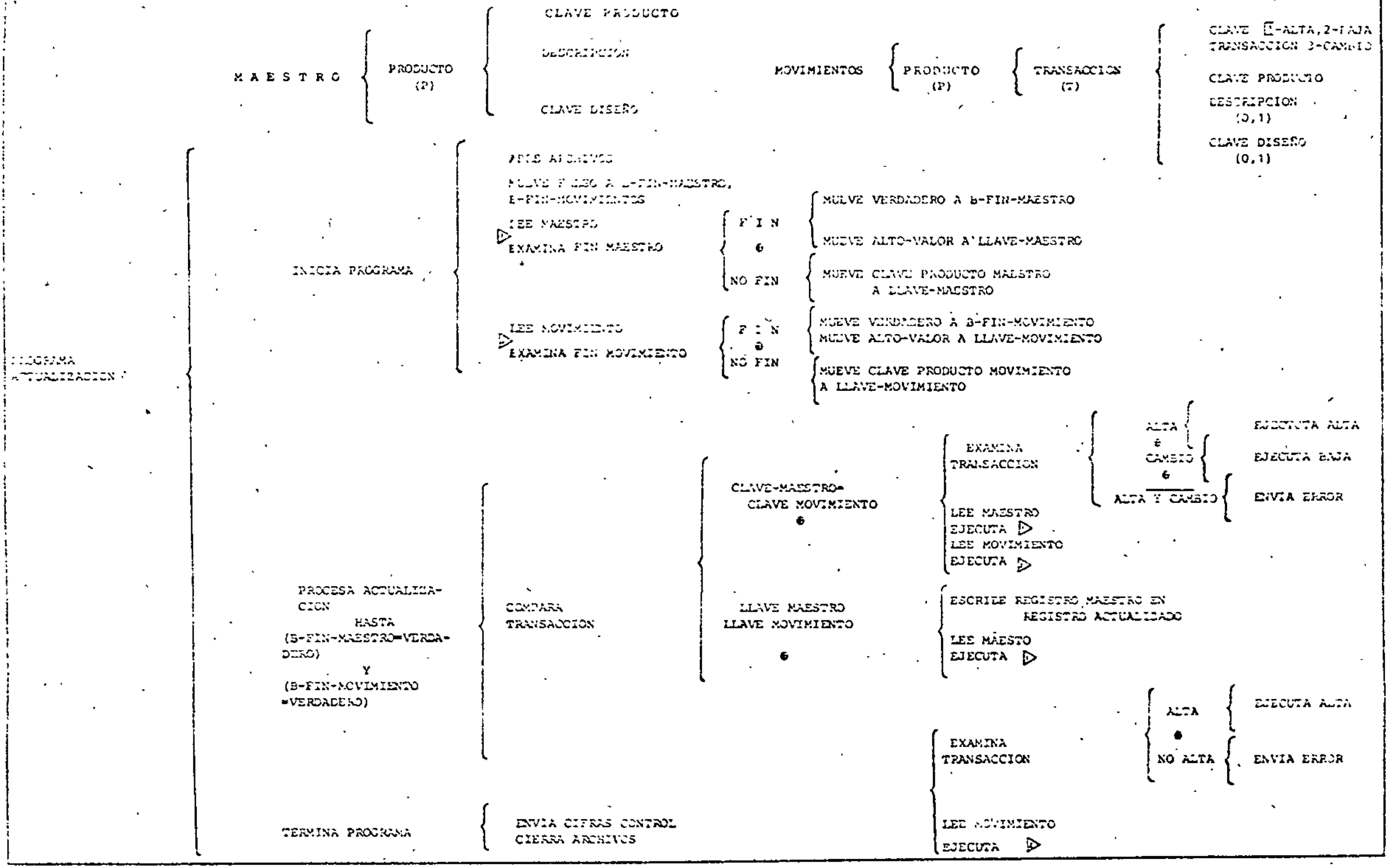


Consideramos que los archivos "Cuestionarios-A" y "Cuestionarios-B", se encuentran previamente clasificados, por lo que es posible empezar por derivar el conjunto programa de mezcla dividiéndolo en inicio, parte media y fin.

```
(ELP) PROGRAMA ENTREGA PEDIDOS
    INICIA PROGRAMA
        ABRE ARCHIVOS
        ESCRIBE "ENTREGA PEDIDOS
        Escribe "FECHA IMPRESION"
        mueve a ceros a
                PEDIDOS EMPRESA
        mueve FALSO a B-FIN
        LEE REGISTRO IDEAL
        EXAMINA-FIN REGISTRO ▷
    PROCESA ESTADO
    HASTA
    (B-FIN = VERDADERO)
        INICIA ESTADO
            ESCRIBE CLAVE ESTADO
            GUARDA CLAVE ESTADO EN
                    W-CLAVE ESTADO
            MUEVE CEROS A
                PEDIDOS ESTADO
        PROCESA CLIENTE
              HASTA
        (B-FIN = VERDADERO)
              O
        (CLAVE ESTADO ≠ W-CLAVE ESTADO)
            INICIA CLIENTE
                    ESCRIBE CLAVE CLIENTE,
                NOMBRE, DIRECCION
                GUARDA CLAVE CLIENTE EN
                  W-CLAVE-CLIENTE
                MUEVE CEROS A
                      PEDIDOS CLIENTE
            PROCESA PEDIDO HASTA
            (B-FIN = VERDADERO) o
            (CLAVE ESTADO ≠ W-CLAVE-
                              ESTADO)
                                 O
            (CLAVE CLIENTE ≠ W-CLAVE-
                              CLIENTE)
                ESCRIBE PEDIDO
                 SUMA 1 A PEDIDOS
                 CLIENTE
                LEE REGISTRO IDEAL
                EXAMINA FIN REGISTR.
                                ▷
                TERMINA CLIENTE
                SUMA PEDIDOS CLIENTE A
                PEDIDOS EMPRESA
                ESCRIBE
                "TOTAL PEDIDOS POR CLIENTE"
                ESCRIBE PEDIDOS CLIENTE
        TERMINA ESTADO
            SUMA PEDIDOS ESTADO A
              PEDIDOS EMPRESA
            ESCRIBE "TOTAL PEDIOS POR
                          ESTADO"
            ESCRIBE "PEDIDOS EMPRESA"
    TERMINA PRO-
    GRAMA
        Escribe "TOTAL PEDIDOS POR
                  ENTREGAR"
        Escribe   " PEDIDOS EMPRESA "
        CIERRA ARCHIVOS

▷ EXAMINA FIN REGISTRO
    FIN
        MUEVE VERDADERO
          A B-FIN.
    ⊕
    NO FIN
        NULO
```

```
PROGRAMA MEZCLA
    INICIA PROGRAMA
        ABRE ARCHIVOS
        MUEVE FALSO A B-FIN-A,B-FIN-B
        LEE "CUESTIONARIOS -A"
        ▷1 EXAMINA FIN CUESTIONARIOS-A
            FIN
                MUEVE VERDADERO A B-FIN-A
                MUEVE ALTO VALOR A LLAVE -A
            ⊕
            NO FIN
                MUEVE ESTADO Y MUNICIPIO A LLAVE-A
        LEE CUESTIONARIOS-B
        ▷2 EXAMINA FIN CUESTIONARIOS-B
            FIN
                MUEVE VERDADERO A B-FIN-B
                MUEVE ALTO-VALOR A LLAVE-B
            ⊕
            NO FIN
                MUEVE ESTADO Y MUNICIPIO A LLAVE-B
    PROCESA CUESTIONARIOS HASTA
    (B-FIN-A=VERDADERO)
        Y
    (B-FIN-B=VERDADERO)
        COMPARA
            LLAVE-A = LLAVE-B
                ESCRIBE REGISTRO-A
                EN REGISTRO-C
                ESCRIBE REGISTRO-B
                EN REGISTRO-C
                LEE CUESTIONARIOS-A
                EJECUTA ▷1
                LEE CUESTIONARIOS-B
                EJECUTA ▷2
            ⊕
            LLAVE-A  LLAVE-B
                ESCRIBE REGISTRO-A
                EN REGISTRO-C
                LEE CUESTIONARIOS-A
                EJECUTA ▷1
            ⊕
            LLAVE-A  LLAVE-B
                ESCRIBE REGISTRO-B
                EN REGISTRO-C
                LEE CUESTIONARIO-B
                EJECUTA ▷2
    TERMINA PROGRAMA
        ENVIA CIFRAS CONTROL
        CIERRA ARCHIVOS
```

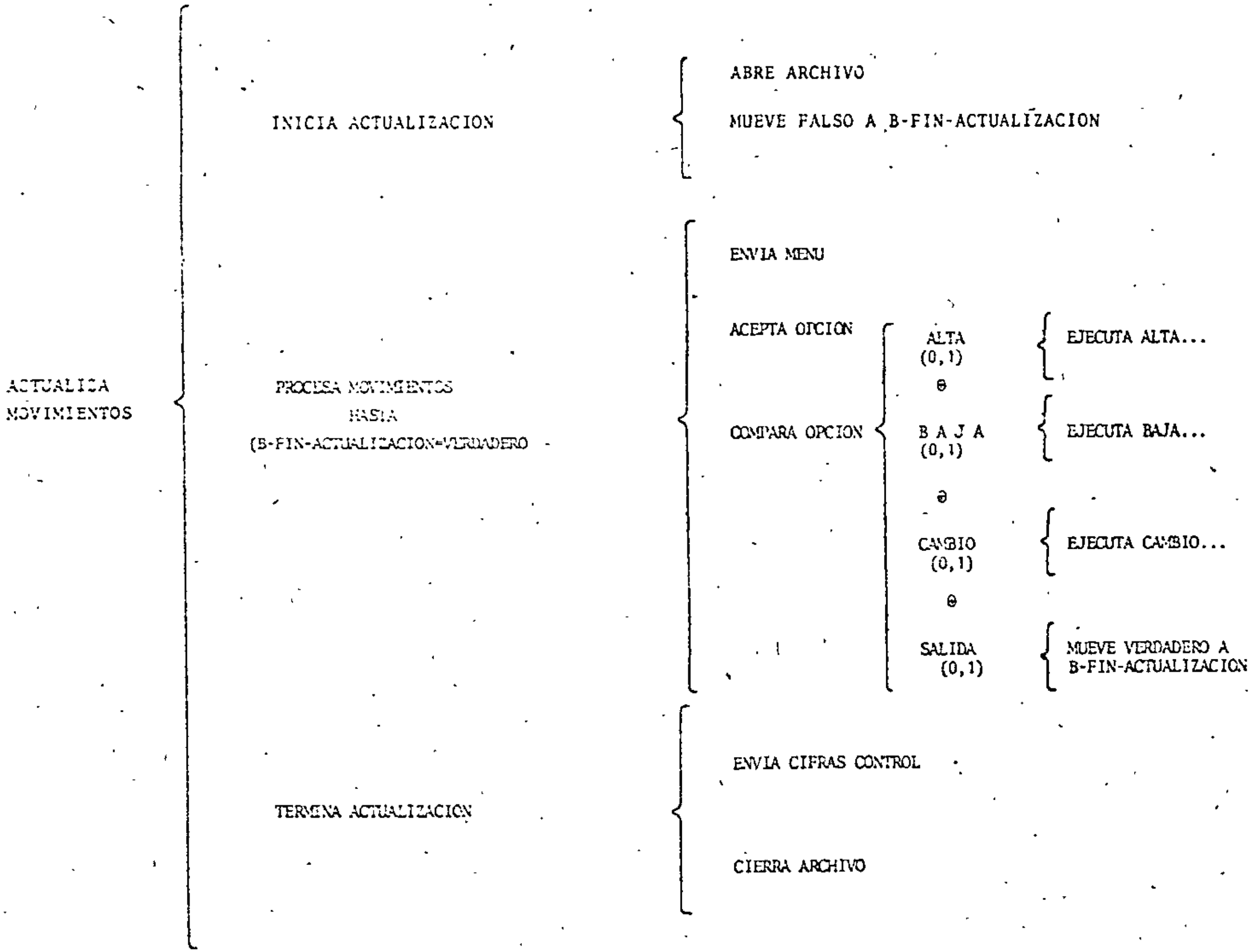
UN [illegible] TIPICO DE LA ACTUALIZACION DE [illegible] "[illegible]", DONDE PREVIAMENTE SE ENCUENTRAN CLASIFICADOS LOS MOVIMIENTOS, PARA AFECTAR MEDIANTE ALTAS, BAJAS Y CAMBIOS A UN ARCHIVO EXISTENTE Y GENERAR UN ARCHIVO ACTUALIZADO, QUE SEAN POR EJEMPLO LOS SIGUIENTES ARCHIVOS IDEALES.



- MAESTRO
  - PRODUCTO (P)
    - CLAVE PRODUCTO
    - DESCRIPCION
    - CLAVE DISEÑO
- MOVIMIENTOS
  - PRODUCTO (P)
    - TRANSACCION (T)
      - CLAVE TRANSACCION 1-ALTA, 2-BAJA 3-CAMBIO
      - CLAVE PRODUCTO
      - DESCRIPCION (0,1)
      - CLAVE DISEÑO (0,1)

- PROGRAMA ACTUALIZACION
  - INICIA PROGRAMA
    - ABRE ARCHIVOS
    - MUEVE FALSO A B-FIN-MAESTRO, B-FIN-MOVIMIENTOS
    - ▷ LEE MAESTRO
    - EXAMINA FIN MAESTRO
      - FIN
        - MUEVE VERDADERO A B-FIN-MAESTRO
        - MUEVE ALTO-VALOR A LLAVE-MAESTRO
      - ⊕
      - NO FIN
        - MUEVE CLAVE PRODUCTO MAESTRO A LLAVE-MAESTRO
    - ▷ LEE MOVIMIENTO
    - EXAMINA FIN MOVIMIENTO
      - FIN
        - MUEVE VERDADERO A B-FIN-MOVIMIENTO
        - MUEVE ALTO-VALOR A LLAVE-MOVIMIENTO
      - ⊕
      - NO FIN
        - MUEVE CLAVE PRODUCTO MOVIMIENTO A LLAVE-MOVIMIENTO
  - PROCESA ACTUALIZACION HASTA (B-FIN-MAESTRO=VERDADERO) Y (B-FIN-MOVIMIENTO =VERDADERO)
    - COMPARA TRANSACCION
      - CLAVE-MAESTRO= CLAVE MOVIMIENTO
        - EXAMINA TRANSACCION
          - ALTA
            - EJECUTA ALTA
          - ⊕
          - CAMBIO
            - EJECUTA BAJA
          - ⊕
          - $\overline{\text{ALTA Y CAMBIO}}$
            - ENVIA ERROR
        - LEE MAESTRO
        - EJECUTA ▷
        - LEE MOVIMIENTO
        - EJECUTA ▷
      - ⊕
      - LLAVE MAESTRO LLAVE MOVIMIENTO
        - ESCRIBE REGISTRO MAESTRO EN REGISTRO ACTUALIZADO
        - LEE MAESTO
        - EJECUTA ▷
      - ⊕
        - EXAMINA TRANSACCION
          - ALTA
            - EJECUTA ALTA
          - ⊕
          - NO ALTA
            - ENVIA ERROR
        - LEE MOVIMIENTO
        - EJECUTA ▷
  - TERMINA PROGRAMA
    - ENVIA CIFRAS CONTROL
    - CIERRA ARCHIVOS

UN DIAGRAMA WARNIER-PARA ACTUALIZACION INTERACTIVA, ES DECIR DONDE SE SOLICITA MEDIANTE UN MENU, UNA OPCION PARA EJECUTAR ALTAS, BAJAS O CAMBIOS SERIA EN SU PARTE INICIAL LO SIGUIENTE:



ACTUALIZA MOVIMIENTOS

INICIA ACTUALIZACION

ABRE ARCHIVO

MUEVE FALSO A B-FIN-ACTUALIZACION

PROCESA MOVIMIENTOS HASTA (B-FIN-ACTUALIZACION=VERDADERO

ENVIA MENU

ACEPTA OPCION

COMPARA OPCION

ALTA (0,1)

EJECUTA ALTA...

⊕

B A J A (0,1)

EJECUTA BAJA...

⊕

CAMBIO (0,1)

EJECUTA CAMBIO...

⊕

SALIDA (0,1)

MUEVE VERDADERO A B-FIN-ACTUALIZACION

TERMINA ACTUALIZACION

ENVIA CIFRAS CONTROL

CIERRA ARCHIVO

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

ARQUITECTURA DEL SISTEMA
(DOCUMENTACION BASICA)

FEBRERO, 1984

vii. ARQUITECTURA DEL SISTEMA (DOCUMENTACION BASICA).

INTRODUCCION

La computadora digital puede llevar a cabo actividades de procesamiento de información, una vez que le sean proporcionadas una serie de instrucciones u ordenes exactas en un lenguaje de programación.

Los lenguajes de computadora son tan estrictos en su construcción que en lugar de afirmar que un programa está escrito en un lenguaje, se señala que esta escrito en código. Los lenguajes naturales son muy flexibles, expresivamente poderosos y por lo tanto pueden ser ambiguos en el significado de sus construcciones. Por otra parte, los lenguajes de computadora o códigos no son tan flexibles y poderosos como los lenguajes naturales, pero si permiten eliminar totalmente las ambigüedades.

El documento final de la fase de definición de requerimientos es el punto de partida para iniciar la fase del diseño de la arquitectura del sistema.

Al proceso de transformar una serie de instrucciones en lenguaje natural a una serie de instrucciones de mediano nivel (pseudo-lenguaje fácilmente convertible a código) se denomina diseño de la arquitectura del sistema.

Durante el diseño de la arquitectura del sistema, el trabajo del diseñador consiste en elaborar un diagrama de ejecución del sistema de procesamiento de datos que se va a implantar en la computadora.

Cuando se diseña la arquitectura de un sistema de procesamiento de datos, se tiene en mente ciertos objetivos referentes al perfeccionamiento de las operaciones y la reducción de los costos. Algunos de esos objetivos son los siguientes:

1. La intención de estandarizar las unidades.
2. Eliminación de funciones innecesarias.
3. Eliminación de reportes, registros y formas innecesarias.
4. Eliminar datos superfluos.
5. Establecer los controles necesarios y eliminar los excesivos.
6. Eliminar la sutileza innecesaria de la calidad de los requerimientos.
7. Eliminación de duplicación de funciones.
8. Eliminar duplicación de objetivos.
9. Eliminación de duplicación de operaciones.
10. Eliminación de duplicación de información y formas.
11. Facilitar el flujo del trabajo y eliminar las altas y bajas del mismo.

ETAPAS DEL DISEÑO DE LA ARQUITECTURA DE SISTEMAS

Para llevar a cabo la traducción de los requerimientos del sistema es necesario dividir la fase de diseño de la arquitectura en cuatro etapas, cada - una de ellas con objetivos específicos y explicadas en detalle a continuación:

- Arquitectura
- Diagrama de estructura
- Diccionario de datos
- Detallado de módulos

ARQUITECTURA

Según el lineamiento fundamental del diseño de programas de arriba hacia abajo, los requerimientos tienen que dividirse en partes que resulten fáciles de entender; por lo tanto, es importante identificar las principales funciones que se encuentren implícitas en los requerimientos a fin de diseñar un programa de computadora para cada una de las funciones. La Arquitectura de programas de computadora identifica esas funciones y las asocia a un programa.

La documentación de esta etapa deberá incluir la lista de los programas a desarrollar y las interfaces en cada uno de ellos.

Existen herramientas, paquetes de computadoras, útiles para el diseño de programas. Entre estos paquetes están el PSL/PSA. Conservar actualizado todo el diseño a través de uno de estos paquetes permite la producción de reportes actualizados del estado del diseño y el acceso en forma interactiva al diseño, facilitando la toma de decisiones.

La parte medular del diseño de la arquitectura es un diagrama de flujo del sistema que indica por medio de gráficos la interacción de las diferentes partes de un sistema.

El diseño de la Arquitectura puede dividirse en las siguientes actividades:

A. Definición de conjuntos de información.

B. Identificación de los procesos o programas a desarrollar.

C. Definición del comportamiento dinámico.

D. Definición de las interacciones organizacionales.

E. Definición de las responsabilidades.

A. DEFINICION DE CONJUNTOS DE INFORMACION.

Los conjuntos de información se clasifican en tres tipos:

1. De entrada

Los conjuntos de datos de entrada son todos aquellos elementos de información necesarios para la correcta operación de la programación y que usualmente son capturados a través de los diferentes dispositivos de lectura de los sistemas de cómputo, como unidades de cinta magnética, terminales de video, lectoras de tarjetas, etc.

2. De salida

Entre los conjuntos de datos de salida se incluye todo tipo de información producida por el sistema como cheques impresos, reportes, sobres con direcciones, gráficas, etc.

3. De base de datos

La base de datos incluye toda la información que es internamente - producida, mantenida y usada por el sistema. Siendo la base de datos, el lugar de concentración de la información que maneja el sistema, su diseño adecuado es uno de los factores más importantes en el desarrollo de un sistema de cómputo. Para mayor claridad en este capítulo se da una descripción de las Bases de Datos.

B. LA IDENTIFICACION DE PROGRAMAS.

Esta actividad consiste en identificar el número de programas o procesos a desarrollar, indicando las interfaces con las que interaccionan. El resultado de la definición de estos programas se puede documentar utilizando diagramas de flujo de información; en ellos se especifica, para cada programa, los conjuntos de información con que interactuará: Las entradas, — las salidas y la base de datos.

También, y como parte del diseño de la arquitectura, se debe incluir- una explicación breve de las funciones de cada programa.

C. EL COMPORTAMIENTO DINAMICO DEL SISTEMA.

Consiste en especificar los eventos y condiciones que afectan al sistema. Los eventos describen hechos que pueden ocurrir durante la operación del sistema. Las condiciones son enunciados que al cambiar su estado de falso a verdadero o viceversa, causan que un evento suceda. Por ejemplo, la condición: "tarjetas listas" especifica que las tarjetas pueden ser procesadas. El evento: "inicio del reporte de contabilidad" inicia o activa el programa que producirá el reporte de contabilidad; es necesario indicar si el evento es periodico o si es producido por algún otro programa o directamente por interacción humana.

D. LAS INTERACCIONES ORGANIZACIONALES.

Las interacciones organizacionales con el sistema, describen las responsabilidades del medio ambiente con respecto al sistema. Se definen quién proporciona la información de entrada y a donde va la salida del sistema; por ejemplo: el departamento de Personal será el responsable de proporcionar la información de los empleados al sistema utilizando los menús apropiados.

E. LAS RESPONSABILIDADES.

La Arquitectura de un sistema deberá incluir la definición de los responsables de diversas actividades del diseño, los nombres de las personas y su responsabilidad. Por ejemplo, se deberá indicar quien es responsable del diseño conceptual de la base de datos, quien es responsable del diseño de cada uno de los programas, etc., con el objeto de poder dirigir preguntas a la persona adecuada.

DIAGRAMA DE ESTRUCTURA

Una vez definido el número de programas que se van a desarrollar y sus interfases asociados, se procede a desarrollar un Diagrama de Estructura para cada programa, llamaremos a cada función con el nombre de módulo. Un diagrama de Estructura debe documentar la división en módulos y la relación entre ellos.

La descomposición de un programa es una aplicación del principio clásico de solución de problemas; divide y conquista. Pero inmediatamente surge la pregunta. Qué tan fina debe ser la subdivisión ?. Es práctica común, documentada en la literatura, que un módulo resulte aproximada-mente de no más de 50 líneas de código ejecutable, ni menos de 30.

Esto quiere decir que un buen diagrama de estructura debe proponer módulos que sean relativamente fáciles de desarrollar. Módulos con - gran cantidad de líneas de código, obscurecen los objetivos y complican el mantenimiento.

Antes de exponer las técnicas para descomponer un programa en módulos, presentamos a continuación una serie de criterios para evaluar si - un diagrama de estructura ha sido bien diseñado, a saber:

A. TAMAÑO

B. ACOPLAMIENTO

C. COHESION

D. PARSIMONIA

A. TAMAÑO

Si algún módulo excede 50 líneas, probablemente debería de descomponerse en más módulos. Sin embargo se puede tener módulos con más de 50 líneas pero con una estructura muy fácil de enternder. Si un módulo es muy pequeño probablemente su función pueda incorporarse a otro. Cabe aclarar que no pueden establecerse normas que definan el tamaño de los módulos. Este debe ser dejar a juicio del diseñador y de los que revisan el diseño.

B. ACOPLAMIENTO

Acoplamiento es la medida de interdependencia entre módulos; alta interdependencia de módulos refleja que los módulos son poco independientes, y por lo tanto, cuando un módulo es modificado, los módulos vecinos también tendrán que cambiar volviendo difícil el mantenimiento de un programa. Un módulo fuertemente acoplado a otros no es transportable, es decir, es más difícil de usar en otros programas ya que, para ser empleado requerirá de cambios en su código.

Un módulo es independiente cuando se puede usar en otros programas sin necesidad de cambiarlo. Un ejemplo clásico de módulos independientes son las subrutinas que existen en la mayoría de los lenguajes de programación como son las rutinas para calcular la raíz cuadrada o el seno de un ángulo, entre otros.

Aún cuando la independencia de los módulos es altamente deseable, a veces no es posible lograrlo debido a la influencia de otros factores en el diseño o por la misma naturaleza de los módulos. Por eso, Yourdon y Constantine han creado una clasificación del acoplamiento entre módulos para poder evaluar el diseño de Diagramas de Estructuras.

1. ACOPLAMIENTO POR ARGUMENTOS

Cuando se tiene este tipo de acoplamiento, la comunicación entre módulos se logra exclusivamente a través de argumentos. Este tipo de acoplamiento es inevitable puesto que casi siempre los módulos tienen argumentos. Sin embargo la idea es mantener el número de argumentos al mínimo posible.

EJEMPLO:

El siguiente módulo tiene cinco argumentos :

```
SUBROUTINE MODO23( A, B, M, N, L )
```

Si el número de argumentos puede ser menor, el acoplamiento de este módulo - con el programa que lo use se reducirá.

2. ACOPLAMIENTO POR ESTAMPADO

Este acoplamiento resulta cuando uno o más módulos usan como argumento la misma estructura de datos. Por ejemplo, se tienen varios módulos: un módulo que calcula impuestos y usa como argumento el registro de un empleado; otro módulo calcula la prima de vacaciones y también usa como argumento el registro del empleado. Cuando la estructura del registro de empleados — cambie, todos los módulos que usan el registro de empleados tendrán que revisarse para posibles cambios.

Por ejemplo, el registro de empleados puede especificarse en el lenguaje --- PL/I como :

```
DECLARE 1 EMPLEADO
```

```
          2    NUMERO CHAR (5)
          2    NOMBRE CHAR (20)
          2    SALARIO FIXED DECIMAL (7,2)
          2    FECHA-INGRESO
                3 MM      FIXED DECIMAL (2)
                3 DD      FIXED DECIMAL (2)
                3 AA      FIXED DECIMAL (2)
```

Los siguientes módulos usan como argumento el registro EMPLEADO de la siguiente forma,

```
MODULO32 : PROCEDURE  (EMPLEADO, IMPUESTO )
MODULO45 : PROCEDURE  ( EMPLEADO, PRIMA )
```

Sin embargo, se observa que MODULO32 sólo necesita el salario y el MODULO45 sólo necesita la fecha, por lo tanto, para evitar acoplamiento por estampado los argumentos de estos módulos deberían de ser como sigue:

```
MODULO32:  PROCEDURE(SALARIO, IMPUESTO )
MODULO45:  PROCEDURE(MM, DD, AA, PRIMA )
```

Para evitar el acoplamiento por estampado se recomienda no pasar como argumento estructuras completas de datos, a menos que el módulo las use completamente.

3. ACOPLAMIENTO POR CONTROL

El acoplamiento por control ocurre cuando un módulo influye en la ejecución de otro módulo a través de un código o de una bandera. Esto quiere decir que el módulo que pasa el control (Padre) dirige la ejecución -- del módulo llamado (hijo). Para evitar este tipo de acoplamiento se tiene que dividir el módulo controlado en varios módulos: uno para cada función. De esta forma, cuando haya cambios en el número de funciones, los cambios - afectarán solo el módulo "Padre" y no los módulos "hijos".

Una forma de acoplamiento hibrido que combina el acoplamiento por control y el acoplamiento por datos se tiene cuando el control es parte del dato. Por ejemplo, el caso del número del empleado, donde los dos primeros dígitos pudierán representar el número del departamento, etc., esto provoca que la estructura de los datos se limite en su evolución y por otro lado, se produce un acoplamiento por estampado junto con un acoplamiento por control.

4. ACOPLAMIENTO POR DATOS EXTERNOS

Este acoplamiento existe cuando dos o más módulos usan la misma - estructura de datos: uno para leer, otro para escribir y otro para modificar una base de datos o un archivo. Un cambio en la estructura de los datos externos va a provocar que el código de los módulos también cambie. El acoplamiento por datos externos es de los más dificiles de evitar. El diseñador busca por un lado, módulos que realicen una sola función; por el otro -

tiene el problema de que esos módulos emplean una estructura común de datos externos. Como no se puede evitar que el acoplamiento por datos externos - se presente, se recomienda tener un buen diccionario de datos para saber -- qué módulos se ven afectados cuando se modifica una de las estructuras de - datos externos.

5. ACOPLAMIENTO POR DATOS GLOBALES

El acoplamiento por datos globales se presenta cuando un grupo de módulos en un programa comparten una área global de memoria, por ejemplo, - cuando se usa el "COMMON" del lenguaje FORTRAN, el "EXTERNAL" de PL/1 y el "DATA DIVISION" del COBOL. Un problema con este tipo de acoplamiento es que todos los módulos tienen que usar los nombres que aparecen en los "COMMONS" y por lo tanto, el uso de éstos módulos es limitado en otros programas. Un cambio en esta región común obliga al menos a re-compilar los módulos.

En FORTRAN por ejemplo, la posición de los datos en el COMMON es importante. Fuerza la aparición de una extraña versión de acoplamiento por estampado haciendo más dificil encontrar errores, ya que los datos en el -- área global no pertenecen a un módulo en particular. En este caso se recomienda evitar el uso de variables globales. Si esto no se puede evitar se sugiere no usar COMMON's "blancos" y emplear COMMON's "etiquetados" con una sola variable por etiqueta. Esto permitirá que el acoplamiento entre módulos se limite a las variables que cada módulo use; por ejemplo, es más conveniente:

```
COMMON / A     / A ( N )
COMMON / B     / B ( M )
COMMON / C     / C ( N )
```

que declarar:

```
COMMON /ABC/  A(N),B(M),C(N)
```

Puesto que los módulos o subrutinas que usan la variable A, solo tienen que declarar "COMMON /A / A(N)", cuando ocurra un cambio en las variables B y C, estos módulos no serán afectados ni tendrán que ser re-compilados.

6. ACOPLAMIENTO POR CONTENIDO.

El acoplamiento por contenido no se puede lograr en lenguajes como el FORTRAN o el COBOL. Solo en lenguajes como el Ensamblador, en donde un módulo puede modificar la secuencia de instrucciones de otro módulo. Si se presenta la necesidad de programar en Ensamblador se recomienda evitar esta práctica.

C. COHESION.

La cohesión, también conocida con los nombres de: robustez, solidez, o coherencia de un módulo, es una medida de la fuerza con que los elementos dentro de un módulo están unidos. Los elementos o componentes de un módulo -

son las instrucciones, los grupos de instrucciones o las llamadas a otros módulos. Entre más robusto es un módulo es más independiente y necesita menos acoplamiento con otros.

1. COHESION POR FUNCION

Un módulo es funcional si todos sus elementos (incluyendo llamadas a - otros módulos) contribuyen a ejecutar una y solo una función. Este es el tipo de cohesión ideal.

2. COHESION POR SECUENCIA

Un módulo tiene rigidez o robustez secuencial cuando sus elementos ejecutan tareas donde los datos que produce una función son la información que recibe la otra. No es el peor tipo de cohesión y responde a una -- forma de pensar natural: "haga ésto, después lo otro, después aquello" sin embargo, "ésto, lo otro, y aquello" representan funciones que pue-- den ser realizadas por módulos funcionales.

Por ejemplo, el módulo siguiente claramente realiza dos funciones que - pueden separarse en dos módulos independientes.

MODULO XYZ

SUMAR TOTAL DE INGRESOS Y

CALCULAR EL IMPUESTO SOBRE LA RENTA

3. COHESION POR COMUNICACION

Esta estructura se presenta cuando un módulo realiza varias funciones que usan la misma estructura de datos. El problema con éste tipo de cohesión, es que puede suceder que una de las funciones se necesite ejecutar en otra parte del programa sin la presencia de las otras funciones. Por lo que, es mejor separar y crear un módulo para cada función.

Por ejemplo, el módulo siguiente contiene dos funciones que actuan sobre una misma estructura de datos, sin embargo, estas funciones pueden realizarse en módulos separados.

MODULO XYZ

FORME EL VECTOR DE PRIORIDADES

ORDENE EL VECTOR DE PRIORIDADES EN FORMA ASCENDENTE

4. COHESION POR PROCEDIMIENTO

Aquí, varias funciones se han ensamblado en un módulo único, porque el control (no los datos como en la cohesión por secuencia) fluye de una función a otra. Nuevamente, lo mejor es separar las funciones en módulos independientes.

Por ejemplo,

MODULO XYZ

ORDENE EL ARCHIVO DE PERSONAL

CALCULE EL TOTAL DEL INVENTARIO

5. COHESION POR COINCIDENCIA TEMPORAL

En este caso, varias funciones aparecen juntas porque sus elementos están relacionados en el tiempo, por ejemplo, las funciones se tienen que ejecutar al principio o al final de un proceso. Sin embargo, estas funciones deben de separarse o ser parte de otras.

Por ejemplo,

MODULO XYZ

INICIE A CEROS LOS APUNTADORES

EMBOBINE LA CINTA

VERIFIQUE LAS BANDERAS DE ESTADO

5. COHESION POR CLASE

Varias tareas son realizadas en un módulo porque se cree que pertenecen a una misma clase o categoría. Por ejemplo, un módulo de entrada y salida que puede ejecutarse todas las funciones de entrada y salida de un programa es un módulo con poca cohesión. Por el contrario, un módulo robusto es muy específico, realiza solo una función.

6. COHESION POR COINCIDENCIA

Esta es la peor de las cohesiones; simplemente se encuentran varias - funciones en un módulo sin justificación alguna.

D. PARSIMONIA

Este criterio se usa para evitar que se diseñen módulos de propósito general. Cuando se diseñan, sobre todo programas grandes, las funciones de los módulos, deberán ejecutar solo lo necesario para cumplir con los Requerimientos del Programa de Cómputo. Las tendencias de algunos diseñadores a concebir módulos con capacidades más amplias de lo necesario a fin de atender futuros requerimientos deben de manejarse con reservas, ya que diseñar funciones generalizadas pueden elevar el costo del proyecto y retrasarlo sin ninguna necesidad.

DESCOMPOSICION FUNCIONAL

La técnica de descomposición funcional es la forma más simple de - usar el principio de "divide y conquista", y consiste en llevar a cabo, recurrentemente, los siguientes pasos:

1. Redactar en lenguaje natural una instrucción que describa la función a desarrollar.

2. Si la función se puede desarrollar en una página (aproximadamente 50 - líneas de código ejecutable), terminar la descomposición de esta función. En caso contrario, dividir la función en subfunciones y verificar que el conjunto de estas subfunciones es equivalente a la función original.

3. Repetir para cada subfunción el proceso de descomposición.

El diseñador debe de pretender que cada vez que escribe una instrucción en lenguaje natural, existe una computadora que es capaz de entender y - ejecutar la instrucción. Este proceso se conoce también con el nombre de refinamiento progresivo. El diseñador empieza escribiendo una instrucción a -- muy alto nivel. Estas órdenes o instrucciones son escuetas, claras y con mucho contenido. Este contenido es refinado o aclarado por el diseñador al tratar de traducirlo a lenguaje de computadora. Si una instrucción es el nivel X es fácilmente traducible a código, el proceso de refinamiento termina; sino, deberán escribirse, otras instrucciones que realicen subfunciones en el nivel X+1. Las funciones en el nivel X+1 representan subfunciones del nivel X que -

aclaran o hacen más fácil la traducción a código.

El método de descomposición funcional tiene como objetivo dividir - el problema de tal forma que los bloques o módulos de los niveles más bajos - sean fácilmente traducibles a código. La desventaja radica en que la descomposición funcional es realizada casi exclusivamente utilizando como criterio el tamaño de los módulos y la capacidad del diseñador para identificar funciones que sean independientes, pero que integradas cumplan la función del programa. A continuación se describen tres técnicas para realizar la descomposición funcional.

1. DESCOMPOSICION USANDO FLUJO DE DATOS.

El método es también conocido con los nombres de "diseño centrado - en la transformación", y "diseño compuesto", en realidad, no es otra cosa que una descomposición funcional con respecto al flujo de datos. Cada bloque del diagrama de estructura se obtiene usando repetidamente el concepto de "caja - negra" que transforma un conjunto de datos de entrada en un conjunto de datos de salida.

El método consiste en dibujar un diagrama de Flujo de Datos (figura 1) y en base a éste construir el diagrama de estructura. Como toda técnica de diseño, el proceso se repite hasta que los módulos resultlantes sean - fáciles de traducir a código.

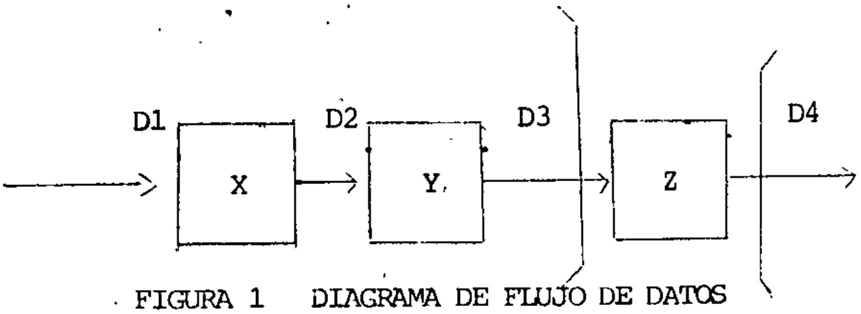


FIGURA 1 DIAGRAMA DE FLUJO DE DATOS

En un diagrama de flujo de datos un bloque representa un proceso de transformación y una flecha representa un conjunto de datos que entra - para ser transformado o que sale ya procesado. Si el diagrama de flujo de datos se usará como diagrama de estructura de módulos, el resultado sería una red de módulos en lugar de una Jerarquía de módulos. Para convertir - el Diagrama de Flujo de Datos (DFD) en un diagrama de estructura jerárquico se selecciona uno o más de los bloques centrales. A continuación se -- "levantan" los bloques seleccionados dejando los bloques de los extremos - "colgando", de esta forma el (DFD) queda dividido en tres partes que corresponden a los tres módulos del primer nivel del diagrama de estructura:

1. El módulo de entrada o lectura, llamado módulo aferente
2. El módulo de transformación central y
3. El módulo de salida o escritura, llamado módulo eferente

Por ejemplo, si en la figura 1 se toma como módulo de transformación central el bloque "Z", entonces, el primer nivel del diagrama de estructura quedará definido como se muestra en la figura 2. El módulo 1.1,- que obtiene el conjunto de datos D3 para ser transformado por el módulo -- central en el conjunto de datos D4, tendrá que realizar las transformaciones "X" y "Y". Esto se logra con dos módulos en el segundo nivel: uno que obtiene los datos D2 y otro que los transforma en D3.

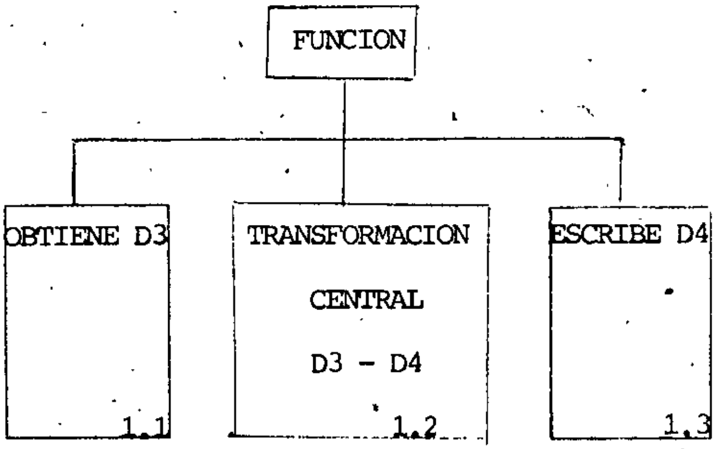


FIGURA 2 PRIMER NIVEL DEL DISEÑO DEL DIAGRAMA DE ESTRUCTURA A PARTIR DEL DIAGRAMA DE FLUJO DE DATOS

Ahora bien, el módulo 1.1.1 de la Figura 3, tiene como objetivo producir los datos D2 para que el módulo 1.1 llame al módulo 1.1.2 y éste los - transforme en D3. Entonces, se necesitan dos funciones en el tercer nivel jerárquico; una que obtenga los datos D1 y otra que transforme D1 en D2.

2. DESCOMPOSICION USANDO ESTRUCTURA DE DATOS.

Dos técnicas de descomposición similares fueron desarrolladas en - forma independiente y practicamente al mismo tiempo por Jackson en Inglaterra y Warnier en Francia.

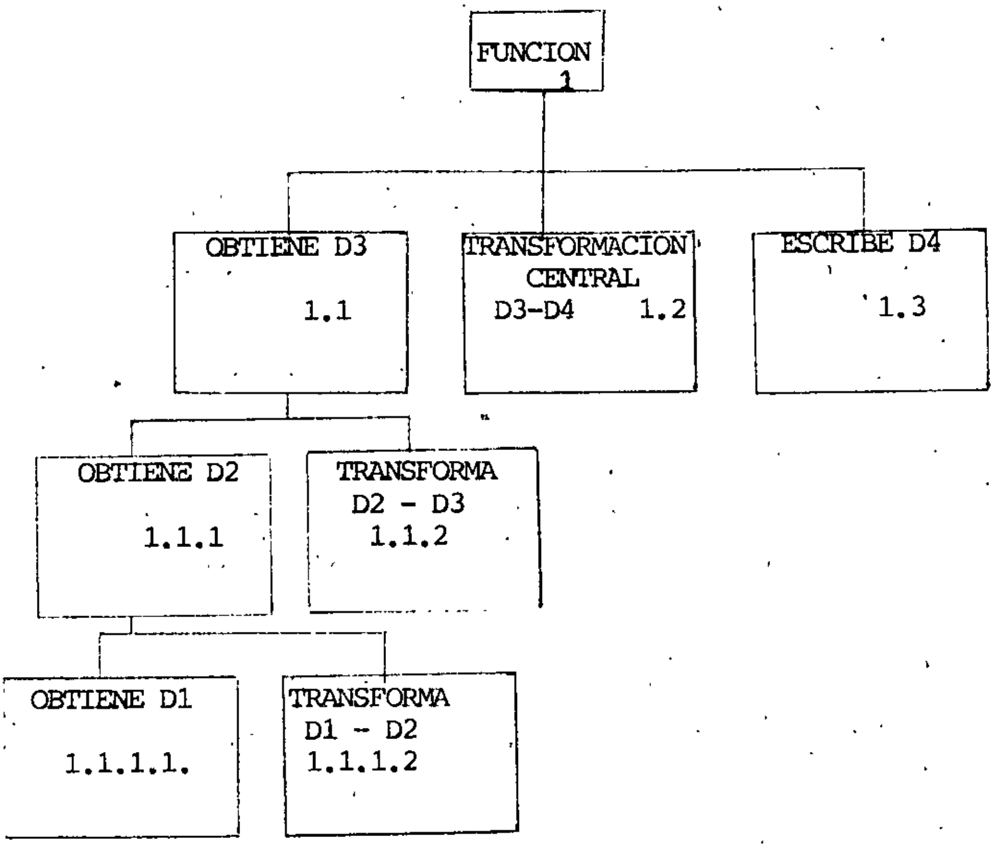


FIGURA 3 DIAGRAMA DE ESTRUCTURA DE DISEÑADO A PARTIR DÉL DIAGRAMA DE FLUJO DE DATOS

Estas técnicas están basadas en el principio de correspondencia - entre la estructura del programa y la del problema que el programa pretende resolver. Como generalmente el problema a resolver consiste en el procesamiento de datos, la estructura del programa deberá corresponder a la estructura de los datos que procesa. Jackson dice que "si un programa no corresponde al ambiente del problema, el programa no es ni pobre ni malo: es incorrecto".

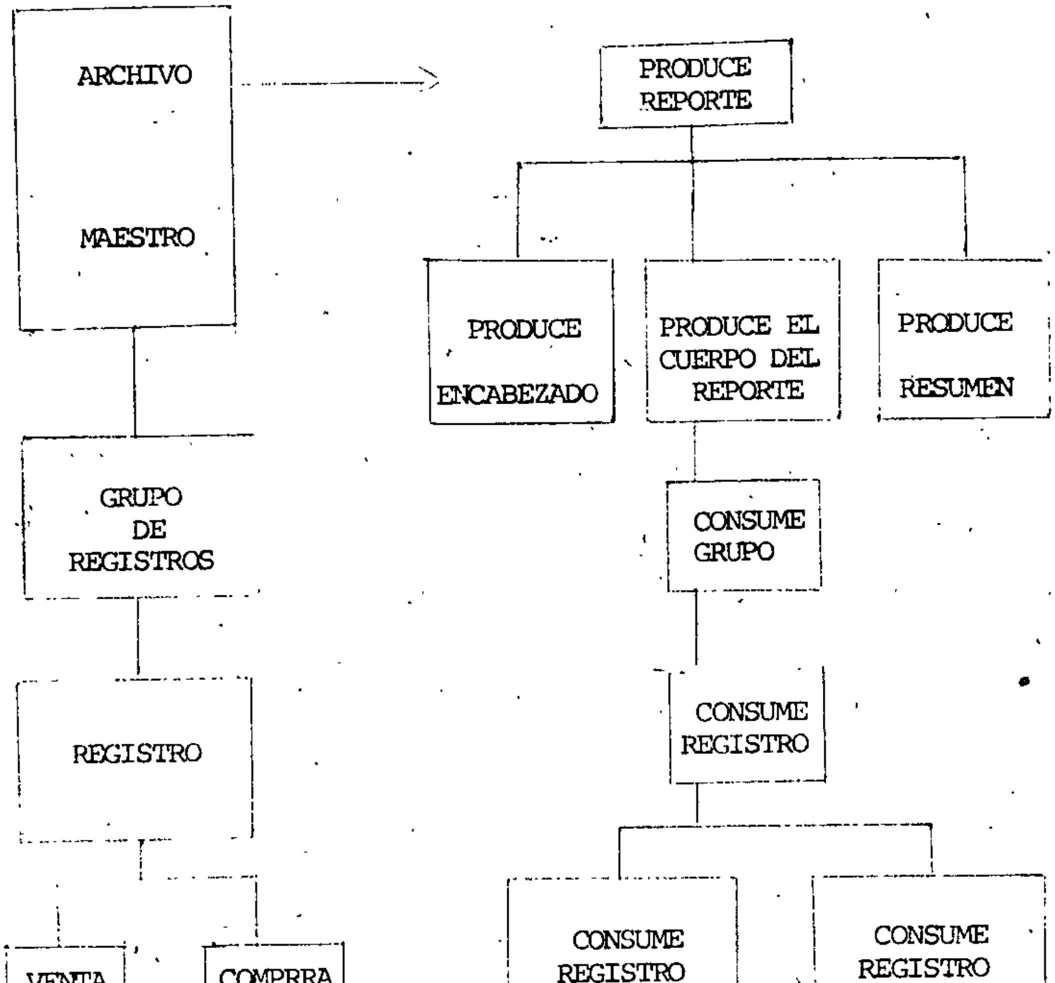
Ejemplo:

En un almacén de una fábrica se requiere producir un reporte que presente el movimiento neto de productos que el almacén compra y vende. Se cuenta con un archivo maestro de transacciones donde se registra cada venta y cada compra. Los registros del archivo contienen, entre otras variables, el número del artículo, el tipo de movimiento (venta o compra) y la cantidad. El archivo se encuentra clasificado por número de artículo de tal forma que todos los registros que representan los movimientos que ha tenido un artículo se encuentran secuencialmente ligados. El programa deberá producir el movimiento neto de cada artículo y al final deberá resumir el total de artículos que sufrieron movimientos. El reporte deberá tener el siguiente formato.

```
                    TITULO


A5/13672     MOVIMIENTO NETO =  -490

A5/17924     MOVIMIENTO NETO =  1500

B31/8200     MOVIMIENTO NETO =   123

--------------------------------------

TOTAL DE ARTICULOS CON MOVIMIENTO = 3
```

La estructura del programa deberá derivarse de la estructura de los datos. Puesto que se desea obtener una estructura Jerárquica, los datos serán analizados jerárquicamente. El diagrama de estructura de datos, así como el diagrama de estructura del programa representan la relación "es compuesto de". Por ejemplo, como entrada al programa tenemos el archivo de transacciones de "es compuesto de" grupos de registros y cada grupo de registro es compuesto de registros individualrs y cada registro es de uno de dos tipos: venta o compra.

(Ver figura 4).



(A)
DIAGRAMA DE ESTRUCTURA
DE DATOS

(B)
DIAGRAMA DE ESTRUCTURA
DEL PROGRAMA

FIGURA 4 DISEÑO DEL DIAGRAMA DE ESTRUCTURA DEL PROGRAMA A PARTIR DEL DIAGRAMA DE ESTRUCTURA DE DATOS.

Por otro lado, a la salida del programa tenemos que la estructura del reporte es la siguiente: el reporte es compuesto de un encabezado, un cuerpo y el resumen. La estructura del programa se deriva de la estructura del reporte y de la estructura de los datos del programa: el programa que produce el - reporte "es compuesto de" tres módulos que produce el encabezado, el cuerpo y el resumen. A su vez, el módulo que produce el cuerpo contiene un módulo que consume un grupo de registros requiriendo a su vez de un módulo que consume un registro y como los registros son de dos tipos se necesitarán dos - módulos: uno para consumir registros de tipo venta y otro para consumir registros de tipo compra. (ver figura 4 (B).

Como puede verse en la figura 4 una parte del diagrama de estructura del programa que produce el reporte correspondiente a la estructura de los datos de entrada y otra parte corresponde a la estructura de los datos de salida.

¿QUE ES UNA BASE DE DATOS?

En esencia todo lo que es -Base de Datos- es la materia prima, los datos, que necesitan las aplicaciones para proveer la información que va a satisfacer las necesidades de los usuarios. Igual que el departamento de producción utiliza a la computadora para controlar la manufactura, Procesamiento de Datos utiliza la Base de Datos para controlar el funcionamiento de las aplicaciones. Para automatizar y controlar el proceso de manejar información, un Diccionario de Datos es necesario. Este y la Base de Datos son herramientas para automatizar las actividades del Departamento de Procesamiento de Datos. Con estas herramientas puede más fácilmente lograr su objetivo apoyar a la organización en su operación diaria.

La información que maneja la computadora no es diferente de la demás información. La Base de Datos es sólo un sofisticado archivo central. Toda la información que se necesita se mantiene, podemos imaginar, en un banco central de archiveros que se organiza en una forma lógica evitando repetir información. No es fácil lograr esto con archiveros, pero debido a que la información está centralizada y todo el mundo lo sabe, es más fácil encontrar lo que cada quien necesita. En lugar de tener cientos de archiveros dispersos por toda la organización con información duplicada ahora la tenemos en un solo lugar.

¿QUE ES UN SISTEMA DE MANEJO DE BASE DE DATOS?

Ahora, ¿por qué necesitamos un Sistema de Manejo de Base de Datos -

(DBMS)? Por la misma razón que nos obliga a tener un empleado que administre nuestro sistema central de archiveros. Naturalmente las fuentes centrales de información requieren un gran número de archivo que deben estructurarse y administrarse en diferentes folders, diferentes cajones y diferentes - archiveros. Algunos de los cajones y archiveros pueden ser utilizados por - cualquier persona. Otros sólo por algunas. Si nada cambiara, nuestro empleado encargado de los archivos tendría poco trabajo. En la práctica, la información es dinámica y también lo son las necesidades de usarla. Dos personas distintas en el mismo puesto de una organización requieren información - distinta para tomar una misma decisión. Es posible que la estructura deba - modificarse para manejar la información de una nueva subsidiaria o para in— cluir una modificación legal.

En este sentido, el DBMS actúa como nuestro empleado de archivo, - él es quien conoce cómo luce la Base de Datos y sabe encontrar la información. Si en algún momento algo cambia el DBMS es el único que debe saberlo. Los - usuarios de la información no tienen por qué enterarse, solamente piden la - información y nuestro empleado se las proporciona. De esta forma la DBMS -- actúa como intermediario entre la información en bruto -la Base de Datos_ y las aplicaciones y usuarios de la información. En el fondo hace independientes a las aplicaciones de la forma en que los datos estén estructurados.

¿QUE O QUIEN ES EL ADMINISTRADOR DE BASE DE DATOS?

Nuestro empleado conoce dónde están todos los archivos y sabe quién puede usar qué. Este empleado probablemente organizará sus archivos para satisfacer sus propias necesidades e inevitablemente la organización de los archivos no podrá satisfacer todas las posibles necesidades. Alguien tiene que supervisar al empleado de archivo y encontrar cuál es la mejor.

Esta función, cuando hablamos de Base de Datos, forma de estructurar la información para cubrir las necesidades de la empresa, se conoce como Administración de Base de Datos y su alcance varía de empresa a empresa. En algunos casos es cubierta por un individuo y en otros por un departamento completo. En términos generales el Administrador de Base de Datos es la persona que entiende qué hay en los archivos y conoce las necesidades de información. El es quien supervisa al empleado - o al DBMS- y cuida que los archivos -o la Base de Datos- contenga la información que se requiere.

Claro que el trabajo del Administrador puede ser muy completo. En aplicaciones con Base de Datos puede haber mucha información, posiblemente miles de millones de caracteres. El Administrador también necesita herramientas que le ayuden a manejar toda esa información. Necesita un índice, un registro. Necesita un Diccionario de Datos.

EL DICCIONARIO DE DATOS INTEGRADO.

Muchos intentos por implantar Base de Datos han fallado porque no contaron con las facilidades que da un Diccionario de Datos. No podemos decir que una Base de Datos es sencilla, tiene que ser compleja si queremos que refleje la información que maneja una organización. Los humanos no podemos manejar esta complejidad sin un conocimiento detallado. El Diccionario existe para almacenar el conocimiento que hay en la Base de Datos y va más allá.

Ofrece un medio sencillo para el Administrador y el departamento de Procesamiento de Datos para documentar todos los aspectos de todos los sistemas quiénes son los usuarios, qué información utilizan, qué hacen con ella, qué información pueden usar y cómo afecta esto a otros usuarios. En pocas palabras el Diccionario de Datos es la Base de Datos del departamento de Procesamiento de datos.

A nivel elemental el Diccionario es una fuente de documentación para el departamento de Procesamiento de Datos, pero también ofrece un medio para controlar el desarrollo de aplicaciones para estar seguros de que nadie reinventa la rueda. Veamos algunas de las cosas que puede contener el Diccionario. Procesar los datos significa simplemente controlar y mover datos elementales que existen en diversos archivos. Para hacer esto, hay muchas cosas que debemos controlar. Por ejemplo, el Diccionario maneja sinónimos, ésto significa que un dato puede tener varios nombres. Un usuario que ha llamado al número de parte "Número de Parte" por veinte años es difícil que cambie y comience a llamarlo "Código de Parte", solamente porque el Procesamiento de Datos se lo pide. Hay que recordar que la función de un sistema de cómputo es servir a los usuarios, por lo tanto debemos contar con los mecanismos que nos -- ayuden a hacerlo, satisfaciendo al mismo tiempo los requerimientos de la computadora. El Diccionario puede usarse para documentar los distintos nombres de los datos y así asegurar que cualquier referencia a alguno de esos nombres encontrará la información correcta.

La validación de los datos es otra tarea que es más fácil de controlar si se cuenta con herramientas automáticas. Es usual que cada dato siga -

reglas específicas de estructura. Posiblemente un cierto dato debe empezar con la letra B y tener siete caracteres de longitud con un campo numérico en cierta posición. Para controlar el desarrollo de aplicaciones, la validación de datos debe ser rígida para así evitar la corrupción de los datos en la Base de Datos.

Además, existe el delicado punto de la privacia. ¿Quién puede usar cada dato? El Administrador de Datos debe saber quién y en qué forma puede hacerlo; qué reportes deben presentarse en una pantalla de rayos catódicos y cuáles pueden imprimirse más tarde en papel. Otro factor son los requerimientos de auditoría, ¿existe algún dato que requiera un tratamiento especial por este motivo?

¿ Y qué hay respecto a la descripción de los archivos? ¿Cuál es su estructura en COBOL? Un dato. ¿existe en la Base de Datos o en un archivo convencional? ¿Qué papel juega cada dato en los programas que lo usan? Para controlar la producción de aplicaciones, el Administrador de Datos debe manejar estos aspectos y muchos más. Para hacerlo debe usar un Diccionario.

Pensando en las facilidades que debe ofrecer un buen Diccionario. CULLINANE desarrolló el Diccionario integrado de Datos (IDD), para documentar completamente la función del departamento de Procesamiento de Datos. En él se describe cada faceta del ambiente de computación sistemas, programas, archivos, quiénes usan estas facilidades, y respecto al teleproceso, cuál es la configuración de la red, cuáles son las alternativas ante alguna falla en la misma, cómo debe presentarse la información en las pantallas, etc. En el IDD es posible definir todo ésto y complejas interrelaciones.

Contando con esta fuente de conocimiento, necesitamos formas de extraer la información y analizarla, obteniendo respuestas para preguntas tales como ¿Cuáles usuarios utilizan el "Número de Parte"? ¿Qué aplicaciones deben ejecutarse el lunes por la mañana?. La documentación que entrega el Diccionario, es parte de la automatización de la función de Procesamiento de Datos.

¿CUAL ES LA MEJOR FORMA DE DESARROLLAR APLICACIONES?

Tener una Base de Datos y un Diccionario es una cosa, pero no sirve para nada si sucede que cuando el usuario va al departamento de Procesamiento de Datos con una petición de información le dicen que tiene que esperar seis meses. La Base de Datos debe contar con mecanismos que faciliten - el desarrollo de aplicaciones que satisfagan las necesidades de los usuarios.

El más elemental de estos mecanismos es el Lenguaje de Manejo de - Datos (DML). Este permite desarrollar aplicaciones usando Lenguajes de programación convencionales, como COBOL, FORTRAN, etc. Un Diccionario es tan - bueno como su capacidad de mantenerse actualizado; por esta razón, el DBMS requiere un Diccionario que se actualice automáticamente. En el IDD de - - CULLINANE, la actualización automática del Diccionario es un subproducto del DML.

El DML cubre dos objetivos. Primero, mantiene actualizado el Diccionario debido a que cuando se utiliza el DML para escribir aplicaciones, el IDD automáticamente captará las estadísticas que se requieren para actualizarlo. Así sabrá, que aplicaciones usan, qué datos, y qué tipo de proceso -

se está realizando. En segundo lugar, DML permite a los programadores escribir aplicaciones más rápidamente. Es una forma más fácil de dar instrucciones, y debido a que el IDD conoce cómo lucen los archivos, puede entregar su descripción al programador evitándole así, la necesidad de conocer y describir detalles de los datos que va a procesar.

Sin el Diccionario, controlar el Procesamiento de Datos es una tarea titánica y la más de las veces imposible. El IDD describe además, cómo - luce la Base de Datos, y todos los accesos a ella son controlados a través — del Diccionario, que es una herramienta interactiva.

En él se describe quién utiliza la Base de Datos, el IDD, y en qué forma; además permite definir los requerimientos propios de la instalación.- En adición puede ser usado como una herramienta de análisis para efectuar revisiones de "¿Qué pasa si?". Estas son vitales para el desarrollo de nuevas aplicaciones así como el mantenimiento a las ya existentes. Por ejemplo, si el número de parte es de seis dígitos, y por alguna razón tiene que ser ampliado, se debe de poder medir el impacto que dicho cambio implicará. Si el DBMS está realizando su trabajo adecuadamente, el cambio será intrascendente para las aplicaciones individuales, pero de todas formas existen algunas implicaciones relacionadas con el usuario: por ejemplo, "¿Quién se va a ver — afectado si se cambia el número de parte?", "¿Quién usa ese dato?", "¿Quién es responsable de actualizarlo?", "¿En cuántas aplicaciones diferentes se — procesa?", etc.

Lo que está sucediendo es que la Base de Datos está trabajando con el Diccionario, y dando todo el control al Administrador de la Base de Datos. Al usar el IDD, todo el departamento de Procesamiento de Datos usa información actualizada. Los cambios se efectúan de una manera ordenada y efectiva.

El verdadero objetivo del departamento de Procesamiento de Datos - satisfacer las peticiones de los usuarios finales- puede ser alcanzado más - rápidamente. El Administrador de la Base de Datos, con un buen DBMS y un buen Diccionario, tiene las herramientas que le permiten no sólo escribir aplicaciones, sino proveer a los usuarios de aquellas aplicaciones que satisfacen sus objetivos básicos.

Uno de los mayores problemas a que nos enfrentamos hoy en día, es cómo reducir el costo y el ciclo del desarrollo de una aplicación usando para ello técnicas menos sofisticadas. Cuántas veces no sucede que un usuario va con el Gerente de Procesamiento de Datos sólo para encontrar que la respuesta que está buscando puede tardar ocho semanas, porque no existe suficiente personal para escribir programas COBOL. Si queremos solucionar el problema, debemos encontrar una mejor forma de desarrollar aplicaciones.

Debemos dejar atrás los Lenguajes de programación laboriosos, e ir hacia herramientas controladas e impulsadas por el Diccionario de Datos. El tipo de herramientas puede estar relacionado directamente con el tipo de problema. Tómese por ejemplo la simple petición del director financiero, que - quiere saber "Quién fué el mejor cliente de la compañía el mes pasado". El necesita un lenguaje de consulta, una facilidad que le permita hacer preguntas, simples, directas y no planeadas. La información está en la Base de Datos, y - no hay razón por la cual alguien que necesita una respuesta rápida deba esperar tres semanas hasta que alguien escriba un programa para él.

CULLINANE provee la solución a esta demanda a través de su facilidad de Consulta en Línea (OLQ). Al mismo tiempo existen sistemas de aplicación -- batch. Por ejemplo, un banco necesita producir estados de cuenta mensualmente, independientemente de que el gerente utilice OLQ para consultar el saldo de un cliente en particular. Los generadores de salidas, como el CULPRIT de CULLINANE, puede generar reportes impresos de este tipo. Para necesidades más complejas, como los sistemas en Línea que actualizan a la Base de Datos, se requiere aún - más flexibilidad.

Idealmente lo que realmente se necesita es un generador de aplicaciones, una herramienta que nos permita desarrollar aplicaciones sin necesidad de escribir programas. Suena como ficción, pero no lo es. Tenemos todos los ingredientes necesarios. Si actualmente podemos definir los requerimientos del - usuario en el Diccionario, ¿por qué no formar a partir de ellos las aplicaciones? ¿Por qué no habría de existir un programa generalizado y único que busque en el Diccionario, localice lo que necesita, y simplemente se ejecute?

Ha habido varios intentos de resolver el problema. La mayoría de - - ellos simplemente generan programas. Pero de lo que CULLINANE está hablando - con su producto ADS/OnLine, es que ya no será necesario generar aplicaciones.

La mayoría de los usuarios hoy en día trabajan sus aplicaciones desde una pantalla de rayos catódicos. Una serie de pantallas le dan la información y las facilidades necesarias. La primer pantalla, por ejemplo, puede ser una serie de opciones. El usuario presiona un botón, tecla un valor, y se mueve hacia la siguiente pantalla, que puede ser de datos, u otro juego de opciones, y así sucesivamente.

Actualmente es posible describir los formatos de pantalla en el IDD, lo que significa que podemos definir en el IDD las necesidades básicas del usuario. De esta manera, estamos a un solo paso de obtener un generador de aplicaciones completo. Con ADS/OnLine, también se pueden definir diálogos al IDD. Estos son, formatos de pantallas, que tienen un nombre dado, y opciones seleccionables, que guían al usuario a través de los varios procesos. Para las varias respuestas que pueden existir a un formato, es posible definir el siguiente paso a realizar, que puede ser, desplegar otro formato, una actualización a la Base de Datos, o cualquier otra función. De esta forma, con simples diálogos, es posible definir en el IDD, todos los requerimientos del usuario.

Desarrollar aplicaciones es ahora, simplemente reunirse con el usuario y dibujar en la pantalla los formatos que él necesite, definir la secuencia de las funciones que se van a realizar, y así sucesivamente. En esencia, se pueden desarrollar prototipos de lo que el usuario necesita, y permitir que él mismo los pruebe exhaustivamente. Si algo necesita afinarse, es tan simple como actualizar el Diccionario. Al final del proceso, el sistema está listo.

La estructura del departamento de Procesamiento de Datos y su forma de operar, cambian radicalmente con una herramienta de este tipo. El COBOL, - FORTRAN, etc. no morirán de la noche a la mañana. Los programadores seguirán usando estas herramientas para labores muy específicas. ADS/OnLine permite al departamento de Procesamiento de Datos responder rápidamente a las- necesidades de los usuarios.

¿QUE HAY ACERCA DE LOS AUDITORES?

Es importante contar con un generador de aplicaciones, pero no debemos olvidar que cualquier sistema debe ser auditable. Los sistemas que — usan Base de Datos son dificiles de auditar con las técnicas tradicionales.

El auditor debe contar con una herramienta que le permita manejar- con independencia la información de la Base de Datos. El EDP-AUDITOR de - - CULLINANE es esa herramienta.

PARA UNA PREGUNTA SENCILLA, UNA RESPUESTA SENCILLA.

Los usuarios no técnicos también requieren herramientas de consulta. Pensemos en un gerente que no tiene mucho tiempo para aprender la terminología de computación. Para él, un lenguaje complejo que requiera de lógica Booleana no es útil. Es probable que sea más fácil comprar una microcomputadora y manejar su propia información. Imaginemos este fenómeno en cada uno de los departamentos de la empresa, tendríamos un gran número de computadoras incompatibles y totalmente independientes.

Lo que necesitamos en este caso es un lenguaje de consulta extremadamente sencillo que pueda accesar la Base de Datos corporativa. CULLINANE ha desarrollado un sistema que usando tecnología de Inteligencia Artificial permite accesar la Base de Datos con un lenguaje exactamente igual al inglés. Para usar este sistema no se requiere capacitación y cualquier persona puede aprovecharlo.

En el futuro, las pantallas de rayos catódicos serán sustituidas por máquinas que entiendan la voz humana. Un lenguaje de consulta como éste, permitirá que por medio de una llamada telefónica a la computadora obtengamos respuesta a nuestras preguntas.

En cualquier caso, esta clase de facilidades podrán funcionar solamente que podamos resolver el problema del manejo de Base de Datos. Hay aún muchos departamentos de Procesamiento de Datos que tienen el agua al cuello debido a que no pueden responder a las necesidades de los usuarios. Que no tienen control de sus aplicaciones y que cada vez son menos redituables. El objetivo último de las computadoras y del personal que las maneja es colaborar en el logro de los objetivos de la organización. Las Bases de Datos, el Diccionario, los Generadores de Aplicaciones y los Lenguajes de Consulta, son las herramientas para alcanzar dichos objetivos. El único obstáculo para entender la tecnología de Base de Datos es la jerga que se utiliza para describirla. Pero esta jerga está destinada al cesto de la basura.

EL DETALLADO DE MODULOS.

En esta etapa del diseño de un programa de computadora, se procede a la especificación detallada del proceso o función que un módulo realiza. El uso de lenguaje natural para esta especificación podría ser suficiente, pero no cuenta con formas estructurales efectivas para la representación de procesos. Por otro lado, aunque algunos lenguajes de programación tienen las formas estructuradas para la representación de procesos, requieren de un nivel muy bajo de expresión y los conceptos de diseño se perderían en un mar de detalles (especialmente si el lenguaje de codificación es el lenguaje Ensamblador). Además, los lenguajes de programación tienen una sintaxis muy estricta y convenciones que no son necesarias al nivel de diseño.

Por lo tanto, la técnica usada para el detallado de los módulos consiste en especificar el contenido de cada módulo usando un lenguaje intermedio entre el lenguaje natural y el código de programación. A este nivel se requiere usar frases-en lenguaje natural y/o instrucciones semejantes a las del código, enmarcadas en un -pseudo-código- que usa las estructuras de proceso de la programación estructurada: la secuencia, la decisión y la repetición. El objetivo es obtener un nivel de-descripción del proceso tal que, cuando los módulos se codifiquen, una línea de detalle corresponda de una a cinco líneas de código.

En esta etapa deberán especificarse todos los datos con que un módulo interactua. Para ello son necesarios:

1. La especificación de todas las estructuras de datos internos a cada módulo (y las globales a varios módulos, en caso de que se haya decidido usar módulos acoplados por da--tos globales).

2. Las estructuras de los datos externos que ya deberán estar diseñadas, por ser parte del diseño de la Arquitectura.

Dadas las características intermedias del detallado de - procesos, se ha creado un lenguaje para Diseño de Procesos que permite la especificación del contenido de los módulos a base-de lenguaje natural y estructuras de proceso. Sin embargo, el uso del lenguaje natural deberá ser restringido: las frases o-instrucciones en lenguaje natural deberán mencionar datos por-su nombre y ser traducibles a código de una forma trivial.

El lenguaje para diseño de programas consiste en usar el-pseudo-código de estructuras de proceso.

Existen tres estructuras de proceso por medio de las cuales se puede realizar cualquier programa utilizando solamente-estas estructuras lógicas de control simple.

1. La secuencia

2. La decisión o alternativa y

3. La repetición

LA SECUENCIA

La secuencia es una estructura de proceso que consiste en enunciar, una serie de instrucciones una después de otra. Una instrucción es: una frase en lenguaje natural, trivialmente -- traducible a código. Las frases en lenguaje natural son imperativas: idealmente consisten de un verbo activo (calcule, -- lea, escriba, verifique, etc.) seguido de una cláusula objeto- lo más sencilla posible, por ejemplo:

Verifique el crédito del cliente.

Cálcule el deducible del salario.

Encuentre el mínimo de los números en la lista.

Asigne a la variable interés el valor de .15

Llane a la subrutina VERINI

LA DECISION

La decisión es una estructura de proceso que permite especificar alternativas en la ejecución de instrucciones dependiendo de una condición. Por facilidad de exposición se divide la decisión en tres tiros: simple, doble y múltiple, la - forma clásica de representar la decisión es a través de un diagrama de flujo.

- LA DECISION SIMPLE

```
SI condición ENTONCES  instrucción -1
```

Si la condición es verdadera la instrucción -1 se ejecuta, - en caso contrario la instrucción se ignora y el proceso continua.

- LA DECISION DOBLE

```
SI condición ENTONCES
   instrucción - 1
FIN (SI)
   instrucción - 2
```

Aquí se especifica que si la condición es verdadera, la instrucción-1 deberá de realizarse, y si la condición es falsa, entonces la instrucción -2 es la que se ejecuta. Solo una - de las dos instrucciones se ejecuta como resultado de la -- ejecución de la decisión doble.

- LA DECISION MULTIPLE

Esta estructura de proceso es una generalización de la decisión de la decisión doble, la estructura tiene la siguiente- forma:

```
SI        condición-1     ENTONCES
            instrucción-1
SINO SI   condición-2     ENTONCES
            instrucción-2
SINO SI   condición-3     ENTONCES
            instrucción-3
SINO SI   condición-4     ENTONCES
            instrucción-4
       :
SINO
            instrucción-N
FIN(SI)
```

Al igual que en la decisión simple y doble, solamente una de las instrucciones se ejecuta como resultado de la ejecución de la estructura. Por lo tanto, las condiciones deberán ser mutuamente exclusivas. Debido a ésta última propiedad, es común encontrarse como parte del lenguaje de diseño y en algunos lenguajes de computadora la siguiente forma equivalente de la decisión múltiple que se conoce como "CASO":

```
CASO
 condición-1
       instrucción-1
 condición-2
       instrucción-2
 condición-3
       instrucción-3
     .
     .
     .
 SINO
       instrucción-N
 FIN(CASO)
```

Esta especificación de la parte de un proceso puede hacerse usando el caso:

```
CASO
            (mes=1,3,5,7,8,10,12)
            asigne al 31 al número de días
            (mes=4,6,9,11)
            asigne 30 al número de días
```

```
                (año es bisiesto)
                asigne 29 al número de días

SINO
                asigne 28 al número de días
FIN(CASO)
```

Es también, común encontrar la siguiente forma de decisión o alternativa múltiple:

```
CASO            "variable o expresión"
                (Lista de valores-1)
                     instrucción-1
                (Lista de valores-2)
                     instrucción-2
                (Lista de valores-3)
                     instrucción-3
                  .
                  .
                  .
                SINO
                     instrucción-N
FIN(CASO)
```

El significado de esta estructura es intuitivo: si la variable o expresión tiene un valor de la lista de valores-1, la instrucción-1 es ejecutada y así sucesivamente. Si la variable o expresión tiene un valor que no está en alguna de las listas de valores la instrucción-N es ejecutada. por ejemplo:

```
CASO  estado-civil
   (CASADO)
         procese empleado casado
   (SOLTERO)
         procese empleado soltero
   (DIVORCIADO)
         procese empleado divorciado
   (VIUDO)
         procese empleado viudo
   (SEPARADO)
         procese empleado separado
   SINO
         reporte error de estado civil
   FIN(CASO)
```

Es posible que esta estructura de alternativa múltiple se encuentre en algún lenguaje de computadora de alto nivel.

LA REPETICION

La repetición es una estructura de proceso que permite la especificación de la ejecución iterativa de una serie de instrucciones.

Presentaremos los siguientes tipos de estructuras de repetición:

La estructura MIENTRAS
La estructura EJECUTA-HASTA
La estructura REPITE
La estructura CICLO

MIENTRAS

Esta estructura para el detallado de procesos tiene la siguiente forma:

```
MIENTRAS condición
     instrucción(es)
FIN (MIENTRAS)
```

El significado de esta estructura es el siguiente: si la condición es verdadera, la instrucción es ejecutada y después- de ser ejecutada, la condición vuelve a ser evaluada. Si re-- sulta verdadera la instrucción se vuelve a ejecutar. Este proceso continúa hasta que la condición sea falsa, en cuyo caso,- la instrucción no se ejecuta y el proceso ya no se repite.

EJECUTA-HASTA

La estructura del ejecuta-hasta es la siguiente:

```
EJECUTA
    instrucción(s)
HASTA condición
```

Esta estructura de proceso señala que, al ejecutarse la estructura, la instrucción debe de realizarse. Una vez que la instrucción ha sido ejecutada, la condición es evaluada y si su valor es falso, entonces la instrucción deberá ejecutarse de nuevo. Esto se repetirá hasta que la condición sea verdadera. En este momento, la instrucción no se volverá a ejecutar y se dice que la estructura de repetición se ha terminado.

Una diferencia entre la estructura "MIENTRAS" y la estructura "EJECUTA-HASTA" es que la última ejecuta la instrucción al menos una vez.

REPITE

Esta estructura de proceso es una de las estructuras más conocidas para los conocedores del lenguaje Fortran. Tiene la siguiente forma:

```
REPITE variable= E-inició, E-fin-E-incremento
    instrucción
FIN(REPITE)
```

El significado de esta estructura es el siguiente: al empezar la ejecución de la estructura, la variable toma el valor de la expresión E-inicio, si este valor es mayor que el valor de la expresión E-fin, la instrucción no se ejecuta y la estructura se termina. Sin embargo, si el valor de la variable es menor que el valor de E-fin entonces la instrucción se ejecuta. Una vez que la instrucción ha sido ejecutada, la variable se incrementa en un valor igual al de la expresión E-incre mento. Nuevamente si el valor de la variable es menor que el valor de E-fin la instrucción se volverá a ejecutar; en caso contrario, la ejecución de la estructura se termina.

La variable deberá ser de tipo entera, al igual que las expresiones E-fin y E-incremento.

CICLO

La forma de esta estructura es la siguiente:

```
CICLO expresión
     instrucción
FIN(CICLO)
```

Con esta estructura se especifica lo siguiente: al inicio de la ejecución del ciclo, se calcula el valor de la expresión que debe de resultar en un número entero. Este número representa las veces que deberá repetirse la ejecución de la instrucción. Posteriormente la ejecución de la estructura se da por terminada. Si el valor de la expresión es cero o negativo, la instrucción no se ejecuta y se termina la estructura.

A continuación se indica como los conceptos de programación estructurada se usan en el detallado de procesos. Al uso directo de estas estructuras en la forma representada en esta fase mediante palabras reservadas e instrucciones en español se le conoce como especificación de procesos en "pseudo-código". Se entiende por pseudo-código un lenguaje estructurado, más no ejecutable.

Las palabras reservadas son las palabras clave que nos indican la estructura del proceso de un programa, por ejemplo, - SI, ENTONCES, SINO, FIN (SI), etc.

Las estructuras básicas de especificación de proceso forman parte del repertoria de instrucciones de algunos lenguajes de computadora. En este caso, la traducción de un diseño en - pseudo-código al lenguaje de computadora es una tarea trivial.

En la presentación de las estructuras de proceso, se han-usado el concepto de "instrucción" de una manera genérica para indicar, ya sea una frase en lenguaje natural o una estructura de proceso. Cuando se usa una estructura de proceso dentro de otra estructura de proceso, en la parte o partes donde se indica que debe ir una instrucción, se dice que hay un anidamiento, por ejemplo:

```
MIENTRAS condición
    SI condición-2 ENTONCES
        frase en lenguaje natural-1
    SINO
        CICLO expresión
            frase en lenguaje natural-2
        FIN(CICLO)
    FIN SI
          frase en lenguaje natural-3
FIN(MIENTRAS)
```

Note que el anidamiento de estructuras de proceso se representa usando el sangrado de las estructuras y frases en lenguaje natural. Al terminar el detalle de un proceso en la fase de diseño solo se deberá tener frases en lenguaje natural y estructuras de proceso anidadas.

En la práctica, la programación estructurada es una herramienta que permite escribir programas más claramente, más legibles y por lo tanto, con menos errores; sin embargo, no puede- afirmarse que el mero uso de las estructuras de control lleva- natural y automáticamente a escribir programas estructurados;- una serie de reglas mecánicas no puede ser un sustituto de la- claridad del pensamiento.

Desde el punto de vista de la programación estructurada,- la mejor documentación de un programa la constituye la claridad de su estructura; además, la única documentación confiable de un programa es el programa mismo, pues sólo leyendo el código puede el programador dar por hecho lo que hace el programa. De allí el énfasis en la legibilidad del código, base indiscutible de la programación estructurada.

ADMINISTRACION DE PROYECTOS EN INFORMATICA

CONSTRUCCION DE PROGRAMAS

(PAGS. 333-362)

FEBRERO, 1984

CODIFICACION

La etapa de Codificación de programas de computadora tiene como objetivo traducir las especificaciones de proceso de cada módulo, descritas en la fase anterior, en instrucciones ejecutables por un lenguaje de programación específico.

La transformación de la definición del problema en especificaciones de proceso para cada módulo ha sido tratada en las --- secciones anteriores.

La programación o codificación de programas de computadora -- había sido considerada un arte hasta los años 50s , debido básicamente a la falta de métodos diseño de programas. Recientemente, y gracias al trabajo de gentes como Dijkstra, Wirth y Weinbers, se ha empezado a desarrollar metodologías de programación basadas en el método humano de análisis y resolución - de problemas.

Como se vió en la fase de diseño de arquitectura, un problema complejo no puede ser convertido en instrucciones de máquina o código de una manera natural y fácil. Es necesario convertir inicialmente la definición del problema en un lenguaje -- fácil de entender como el "Pseudo-código", y posteriormente -

las operaciones descritas se refinan hasta que finalmente se escribe el código interpretable por máquinas computadoras.

Esta técnica es el corazón de varias metodologías de programación entre ellas, la " Programación estructurada".

El detalle del proceso a realizar, descrito en documentos anteriores, marca la terminación de la fase de arquitectura y es un requisito indispensable para el inicio de esta etapa;- ésta se considera terminada cuando todos los módulos han sido codificados y verificados; sin embargo, las fases de ar-- quitectura y construcción de programas pueden llevarse en paralelo dependiendo del enfoque que se tome.

Existen dos tipos de enfoques al respecto, uno radical y el otro conservador. El radical, se basa en el diseño de uno - de los programas del sistema, procediéndose a su codifica--- ción y prueba inmediatamente. El enfoque conservador, por - su parte, se basa terminar primero en terminar primero el - diseño de todo el sistema, (integrado por diversos progra-- mas de computadora), pasando posteriormente a las etapas de codificación y pruebas para todo el sistema. Desde luego no existe una respuesta absoluta sobre cuál enfoque es mejor. Algunos factores que deben tomarse en cuenta para tomar una -

mejor decisión son :

- CONOCIMIENTO DEL USUARIO

Si el usuario no tiene idea bien clara de lo que quiere y cambia continuamente las especificaciones, use el enfoque radical; en cambio, si el usuario sabe con precisión lo que quiere, intente el diseño completo.

- PRESIONES DE TIEMPO

Si se está bajo presiones de tiempo para producir resultados tangibles rápidamente, use el enfoque radical.

- CALENDARIOS PRECISOS

Si se requiere cumplir con calendarios precisos y detallados de utilización de recursos, deberá optarse por - el enfoque conservador. De otra manera, con el enfoque radical sería muy dificil determinar con precisión el - tiempo de duración y los costos del sistema sin ni siquiera conoce cuantos módulos va a contener el programa.

Independientemente del enfoque elegido, los módulos de un programa deben ser codificados. Existen dos estrategias principales - para codificar estos módulos :

-Codificación descendente (TOP-DOWN PROGRAMMING)

-Codificación ascendente (BOTTOM-UP PROGRAMMING)

Ambas estrategias son factibles, sin embargo la codificación ascendente permite atacar los módulos de los niveles jerárquicos más bajos primero. Estos contienen las operaciones primitivas, y permiten de ahí construir las operaciones más complejas. Una segunda venta de utilizar las codificación ascendente es que la integración de los módulos y los casos de prueba son diseñados- y realizados siguiendo la misma secuencia como lo muestran las- siguientes secciones.

Todos aquellos módulos que han sido probados pasan a formar parte de una herramienta automática conocida como " Biblioteca de Soporte de Programas" (BSP).

Esta biblioteca tiene como función el almacenamiento de todos - aquellos módulos que hayan sido aprobados, y que cumplan con --

los requerimientos del usuario y verificados por control de calidad de acuerdo a las normas establecidas. Estos módulos, de ahora en adelante, se definirán como "Módulos Reusables". La función de verificación por control de calidad se conoce con el nombre de " Auditoría" y se puede realizar con una segunda herramienta automática conocida como "Auditor de Código" .

Los módulos reusables no son los únicos productos finales de esta etapa. Además de estos módulos de códugo fuente insertados en la BSP existen listados de impresora, módulos objeto, procedimientos catalogados, resultados de las pruebas y finalmente, y - en caso necesario, las versiones actualizadas de los documentos anteriores.

Resumiendo, la etapa de codificación no deberá de iniciarse antes de completar la Revisión del Diseño de programa a codificar. Esto, sin embargo, no significa que se necesite tener todo el -- sistema diseñado y revisado, para empezar la codificación. Las normas de codificación y las herramientas automáticas para el -- control de esta programación son indispensables para esta etapa.

Los productos de esta etapa del proceso de programación quedan - plasmadas en la documentación que se obtiene del código fuente - generado.

TECNICAS DE CODIFICACION

El objetivo de las Técnicas es proporcionar una guía para la codificación, integración y verificación de los módulos de un programa.

En esta sección se presentan las técnicas empleadas con mayor - frecuencia en la codificación de programas de computadora.

Las técnicas de codificación tienen como objetivo facilitar la traducción del diseño detallado de los módulos que componen un programa a un lenguaje de programación específico, manteniendo la estructura jerárquica definida en la fase de diseño.

La descripción detallada de los módulos sirve como base para la codificación. Cualquier módulo, aún el más complicado, ha sido detallado utilizando las reglas básicas de programación estructurada: los módulos tienen una sola entrada y una sola salida - y pueden ser facilmente traducidos, revisados y probados en forma independiente.

El detalle de los módulos se traduce en instrucciones de máquina (computadora), dependiendo de las características del pro--

pio lenguaje de programación. Mientras algunos lenguajes incluyen las estructuras de control básicas entre sus instrucciones normales (ALGOL, PASCAL, PL/1, FORTRAN 77), en los lenguajes más populares (COBOL, BASIC, FORTRAN 66), estas facilidades no solo no existen, sino que además presentan algunas dificultades para su implementación.

La situación ideal se presenta cuando la instalación de cómputo cuenta con un lenguaje de programación estructurado, con lo cual la función de traducir el detalle de los módulos se convierte en una tarea trivial como se muestra a continuación :

```
  ESTRUCTURA                             IMPLEMENTACION FORTRAN 77

Mientras (bandera="S")                     WHILE(BANDERA="S")
  Llama actualiza                            CALL ACTUALIZA
  Limpia pantalla                            REWIND 3
  Escribir (pantalla) "cambios (S,N)?"       WRITE(3,5000)'cambios(S,N)?"

  Leer (pantalla) bandera                    READ(3,5100) BANDERA
Fin (mientras)                             END WHILE
```

Sin embargo, esto no sucede comunmente con los lenguajes utilizados en la mayoría de las instalaciones de cómputo. En el caso del lenguaje de programación COBOL, por ejemplo, la proposición, "Si-entonces-sino" no puede ser implementada directamente, pués no permite el uso adecuado de proposiciones unidas como se muestra a continuación. La desventaja de éste hecho es que el código de un módulo no permanece unido, sino que hay que dividirlo en secciones y lugares físicos diferentes.

```
  ESTRUCTURA                        IMPLEMENTACION COBOL

 Si P entonces                      IF P THEN
    Si Q entonces                      PERFORM IF-INTERNO
       Instrucción-1                  INSTRUCCION-3
    Sino                            ELSE
       Instrucción-2                  INSTRUCCION-4
    Fin (si)
    Instrucción-3                  IF INTERNO SECTION.
 Sino                                  IF Q THEN
    Instrucción-4                      INSTRUCCION-1
Fin (s).                            ELSE
                                        INSTRUCCION-2
```

En el caso de los compiladores FORTRAN 1966 (FOR66), el mayor problema consiste en que solo reconocen instrucciones simples y por lo tanto la estructura lógica del programa debe ser construida -- utilizando la instrucción de remificación incondicionada ("GO TO") la cual, si se utiliza indisciplinadamente obscurece la lógica -- del programa. En cualquier lenguaje de programación deberemos -- restringirnos a usar instrucciones que, aunque no sean traducciones directas del pseudocódigo, al menos si sean implementaciones ordenadas o " emulaciones" de las estructuras básicas.

A continuación se presentan dos técnicas de codificación: Las emulaciones y los precompiladores, especialmente útiles cuando se -- utiliza un lenguaje de programación que no cuenta entre sus instrucciones normales: con las estructuras básicas, definidas en la sección de diseño de arquitectura específicamente en el punto de pseudo-código. Para terminar se menciona una tercera técnica que puede aplicarse a cualquier tipo de lenguaje y se conoce como estilo de programación.

EMULACIONES

A continuación se presenta, en FORTRAN66, las emulaciones de las principales estructuras básicas definidas en la programación estructurada.

ESTRUCTURA (SI-ENTONCES-SINO)

```
Si P entonces                     IF (.NOT.P) GO TO N1
   Instrucción-1                     Instrucción-1
                                  GO TO N2
Sino                          N1  CONTINUE
   Instrucción-2                 Instrucción-2
Fin(si)                       N2  CONTINUE
```

ESTRUCTURA(MIENTRAS)

```
Mientras (condición)          NI  IF  (.NOT.condicion) GO TO N2
   Instrucción-1                      Instrucción-1
                                       GO TO N1
Fin(mientras)                  N2  CONTINUE
```

ESTRUCTURA(EJECUTA)

```
Ejecuta                                GO TO N2
                                   N1  IF (condicion) GO TO N3
                                   N2  CONTINUE
   instruccion-1                         Instrucción-1
                                          GO TO N1
Hasta (condición)                  N3  CONTINUE
                                        END IF
```

ESTRUCTURA (REPITE)

```
Repite inicial, final, incr.           INDICE= INICIAL
                                   N1  CONTINUE
   Instrucción-1                         Instrucción-1
                                       INDICE=INDICE+INCR
Fin(repite)                            IF(INDICE.LE.FINAL)GOTO N1
```

Las desventajas de la emulación de las estructuras son:

Primero, que el número de pasos de descomposición necesarios para llevar el planteamiento original de programa a su implementación es mayor, y

Segundo, que la claridad de un programa escrito en pseudocódigo se pierde al traducirlo a código.

Tomando en consideración los puntos expuestos anteriormente, la solución más popular para la emulación de las estructuras básicas es eluso de un precompilador.

PRECOMPILADORES

Un precompilador es un programa de computadora que extiende la sintaxis de un lenguaje de programación específico permitiendo al programador codificar sus programas utilizando las estructuras básicas (si-entonces-sino, mientras, etc.) en combinación - con las instrucciones propias del lenguaje.

El uso de precompiladores presenta varias ventajas sobre el intento de programar en forma estructurada en lenguajes no estructurados:

a. Portabilidad

b. Menores violaciones a las reglas de programación estructurada.

c. Mayor aproximación entre los procesos de diseño y codificación.

d. Mayor confiabilidad.

e. Mayor mantenibilidad.

Algunos de los precompiladores más populares son el META COBOL, WATFOR y HIFTRAN.

En la práctica la programación estructurada es una herramienta que permite escribir programas más claramente concebidos, más legibles y por lo tanto, con menos errores; sin embargo, no puede afirmarse que el mero uso o emulación de las estructuras de control básicas lleve automáticamente a escribir programas estructurados.

ESTILO DEL PROGRAMADOR

La claridad de un programa depende en gran medida del estilo del programador. Si bien los principios de los que constituyen un - buen estilo de programación no pueden tampoco ser expresados como reglas mecánicas de la buena programación, la experiencia nos indica algunas normas que pueden servir como "guia de estilo" y las cuales se detallarán en la sección de Normas de Codificación.

Resumiendo, la mejor documentación de un programa la constituye la claridad de su estructura; además como ya se mencionó la documentación más confiable de un programa es el programa mismo, --- pués solo leyendo el código se puede detectar lo que hace el programa. Por lo tanto se recomienda la selección de lenguajes que estimulen una programación legible y estructurada, evitando la - necesidad de instrucciones de flujo arbitrariamente complicadas.

NORMAS DE CODIFICACION

Los estándares o normas de programación son un conjunto de reglas de estilo que permiten que los programas sean fáciles de leer, revisar, modificar y por lo consiguiente mantener.

Este conjunto de normas generales de programación tiene como objetivo unificar el "estilo de programación", sin limitar la creatividad del programador. Se entiende por estilo, la selección adecuada de hábitos de programación que favorecen la claridad y eficiencia del programa.

El empleo unificado de estos hábitos coadyuva a la producción de buenos programas. Estos deben de trabajar, eso es lo más importante. La velocidad, la utilización del tiempo de máquina, el número de líneas de código y otras variables cuantificables no tienen significado si el programa no trabaja. Podemos tomar en cuenta los siguientes compromisos:

a) El programa debe ser correcto antes de ser rápido.
b) Debe ser factible antes de ser rápido.
c) Debe ser claro antes de ser rápido.
d) Debe tener bajos costos de desarrollo.

e) Debe tener bajos costos de prueba.

f) Debe tener costos mínimos de mantenimiento. Organizaciones que emplean masivamente procesamiento de información gastan entre el 50 y 90% de su presupuesto anual en el mantenimiento de sistemas existentes.

g) Debe ser fácilmente modificable. Dentro del dinámico - ambiente de una empresa moderna, los requerimientos de los programas cambian continuamente, hasta el módulo -- más pequeño. Un buen programa deberá estar diseñado tomando en cuenta las eventuales necesiades de revisión y cambios.

h) Debe tener un diseño poco complicado. Siempre deberá diseñarse un programa con la idea de que alguna otra persona lo va a mantener.

i) Eficiente. Debe recordarse que en general cerca de 50% del tiempo de ejecución de un programa se emplea sola-- mente en el 3% de las instrucciones. Para obtener ma-- yor eficiencia el programa deberá tener una estructura lógica, con énfasis en simplicidad y confiabilidad. Despues de que el programa esté trabajando se recomienda - reescribirlo y optimizar aquellos códigos que consuman más tiempo.

Para lograr programación con estas características, es necesario establecer ciertas reglas para la elaboración de código.

1. Una entrada, una salida, una sola función.

Es conveniente que todo módulo de programación observa una y solo una entrada, una y solo una salida y que realice una sola función.

Algunos lenguajes de programación no ofrecen la facilidad de modularización.. En este caso todas las funciones de un programa se realizan en un sólo módulo tomando el programa complicado y por lo tanto dificilmente mantenible..

2. Convención de Nombramiento.

El objetivo de una convención para nombrar archivos, variables, subrutinas, etc. es producir una documentación de programas consistente. Como es sabido, en cualquier instalación varios analistas, diseñadores y programadores pueden estar trabajando en el desarrollo o mantenimiento de un mismo sistema. Así mismo un sólo analista, diseñador o programador puede estar trabajando en varios subsistemas. Además, varias personas con diferentes formaciones pueden participar en la revisión, modificación o actualización de programación.

Por lo consiguiente, el uso consistente de una técnica de documentación dentro de una instalación mejora notablemente su utilidad, y asegura que los productos sean únicos y por lo tanto fáciles de identificar y catalogar.

Los nombres de símbolos asignados por el programador para representar variables, archivos y nombres de módulos deben de ser representativos del contenido del mismo.

Un método sencillo para asignar nombres es: escribir una frase significativa del contenido de la variable o función con una, dos o más palabras, comprimir la frase eliminando sucesivamente (de derecha a izquierda) las vocales hasta obtener un nombre significativo o "mnemónico" de longitud adecuada. Además, es importante prestar atención en la pronunciación del mnemónico, ya que esto puede mejorar en la comprensión y legibilidad del programa.

Ejemplos:

Pronóstico de carga	- PRNKGA
Apuntador	- APNTDR

3. Longitud de los programas y módulos

El cuerpo de los módulos (programa principal y subrutinas) no deberá exceder de 100 instrucciones ejecutables y excluyendo comentarios explicativos.

4. Asignación y localización de banderas de compilación condicional.

Para la depuración o rastreo del contenido de las variables durante la ejecución del módulo se sugiere el empleo de la compilación condicional. Esta función se lleva a cabo por medio de banderas de control, las cuales son activadas por el compilador.

Toda subrutina o programa principal deberá incluir ayudas permanentes de rastreo. Estas pueden ser clasificadas de acuerdo a los tipos de variables o cualquier otra división.

Ejemplo:

```
C              ----------------------
C              * AYUDAS DE RASTREO*
C              ----------------------
:SKFZ   N

               instrucciones ejecutables

:ESKP
```

en donde : N representa la bandera que se desea activar.

: SKFZ representa la inclusión de la ayuda permanente de rastreo.

: ESKP representa la terminación de la ayuda permanente de rastreo.

5. Definición explícita de variables.

Algunos lenguajes de programación son limitados y no permiten el uso de identificadores (nombres de variables, archivos y modulos en general) de longitud mayor de 2 caracteres. Otros lenguajes tales como Fortran, Cobol, Algol y Pascal permiten el uso de identificadores de longitud variable.

Los nombres de estas variables deberán ser explícitamente declaradas empleando las instrucciones correspondientes a cada compilador. No deberá dejarse que el sistema asuma, como en - Fortran, que las variables que empiezan con las letras A-H y O-A sean del tipo real y las que comienzan con I-N sean de tipo entero. Esta práctica, además de que puede alterar el - significado en el contenido de la variable, reduce en uno el número de caracteres disponibles para el identificador.

Ejemplo FORTRAN:

```
REAL* 6                    AAAAAA,  BBBBBB
INTEGER* 3                 CCCCCC-  DDDDDD,  EEEEEE
DOUBLE PRECISION* 12       FFFFFF
LOGICAL                    GGGGGG,  HHHHHH
CHARACTER* 1               IIIIII,  JJJJJJ,  KKKKKK
```

6. Autodocumentación

Los comentarios sirven para explicar y/o aclarar que sucede en el flujo de control de programa. Existen dos tipos de - comentarios: comentarios de prólogo o encabezado del módulo y comentarios explicativos del cuerpo del módulo.

6.1 Comentarios de Prólogo.

Todo módulo (programa, subrutina o función), debe contener una descripción breve de la función, uso, varia-- bles que maneja y aclaraciones pertinentes sobre el uso del mismo.

Estos comentarios deben aparecer inmediatamente después del nombre del módulo y pueden estar delimitados por un marco de asteríscos.

Los comentarios que se pueden incluir en la carátula de cada módulo, por orden de aparición, son :

a) Programa fuente.

Referencia del área en disco que contiene el módulo en cuestión.

b) Lanzador.

Refencia del área en disco que contiene el conjunto de instrucciones en lenguaje de control del sistema (Job-Control Language), para la ejecución de dicho módulo.

c) Propósito.

Describe en una o dos líneas la función que realiza el módulo y, en caso de ser relevante, indica el método de solución empleado.

d) Referencias

Menciona las referencias en los distintos documentos RPC, DPC, etc., en donde el requerimiento que se está resolviendo por medio de este módulo queda definido.

e) Bibliografía.

Menciona las fuentes bibliográficas utilizadas como base en el desarrollo de este módulo.

f) Limitaciones.

Sirve para indicar en forma breve las restricciones en el volúmen de información y el rango de valores permisibles para ciertas variables.

g) Aclaraciones.

Esta sección sirve para indicar las condiciones de terminación de la ejecución, las condiciones anormales y las suposiciones.

h) Nombre y Fecha de Implementación.

Identifica el autor y la fecha de terminación asociados- a este módulo.

i) Nombre y Fecha de Revision(es)

Identifica los revisores y la fecha en que éstas fueron- realizadas.

6.2 Comentarios Explicativos.

Estos comentarios se insertan en el código a fin de indicar cual es el propósito de cada sección, conjunto de comandos o estructuras lógicas dentro del cuerpo del módulo.

El comentario deberá tener un sangrado igual al del nivel de anidamiento del código que describa. Deberá estar delimitado por líneas punteadas y comenzar y terminar con - un asterísco.

Ejemplo:

```
C     ---------------------------------------
C     * IDENTIFICAR EL NUMERO DE SUBSISTEMA *
C     * ASOCIADO A CADA INTERCAMBIO         *
C     * PROGRAMADO                          *
C     ---------------------------------------
```

7. Identación o Sangrado.

Lo anterior es con el objeto de hacer visible la estructura global del módulo así como el rango de control que tiene cada estructura lógica. Menos de 3 espacios reduce la legibilidad del módulo, y más de tres es innecesario.

Algunos compiladores tienen la opción de indentar automática (por ejemplo FO.I), con lo cual ya no es necesario el sangrado en el código fuente.

Ejemplo:

```
C   ------------------------------
C   * PROCESO PARA IDENTIFICAR EL*
C   * ARCHIVO DE SALIDA SEGUN EL *
C   * MODO DE PROCESO            *
C   ------------------------------
    IF  (MODO.EQ.O) THEN
C   ------------------------------
C   *MODO DE PROCESO BAJO CONTROL*
C   ------------------------------
      LFN = LFSUB1 (SBTEMA)
  ELSE
C   ------------------------------
C   * MODO DE PROCESO MANUAL     *
C   ------------------------------
      LFN = LFMC4 (ARTEMP, CITEMP )
  END IF
```

8. Especificaciones de error de Operaciones I/O Entrada/Salida

Con el fin de codificar módulos que respondan ante un error en dispositivos de entrada/salida, las instrucciones de abrir, cerrar, leer y escribir archivos deberán contener especificadores de error en la operación y variables que contengan el número de error asociado. Estos especificadores deben de ser utilizados en forma organizada como se muestra en el ejemplo a continuación.

Ejemplo:

```
  C    ----------------------------
  C    * ABRIR ARCHIVO DE ENTRADA *
  C    ----------------------------
  C    OPEN(UNIT=IM FILE=E:RISKGA, IOSTAT=IOS, ERR=3000)
  C    ------------------------------------------
  C    * LEER NUMERO DE DIAS A PRONOSTICAR   *
  C    ------------------------------------------
  C    READ(IM,5000, IOSTAT*IOS,ERR=3000) NPRIA
  C    -------------------
  C    *AYUDAS DE RASTREO*
  C    -------------------

:SKFZ 1
    Write(6,5000)"NPDIA=     ", NPDIA
:ESKP
    NPDIA=NPDIA+1
  C    -----------------
  C    * CERRAR ARCHIVO *
  C    -----------------
       CLOSE(IM, STATUS=_' KEEP_',IOSTAT=IOS, ERR=3000)
       IOS=0%/
3000   IF(IOS.EQ. 0%) THEN
  C    -------------------
  C    * ERROR EN ARCHIVO *
  C    -------------------
         WRITE(3,_*) _*ERROR=_',IOS, _ EN LFN= _ ', IM
       END IF


       * Formato de Lectura *


5000   FORMAT (10.2)
       CALL EXIT
```

9. Números Mágicos

Deberá prohibirse el empleo de números mágicos dentro del código del módulo. Estos números son constantes que aparecen en el código sin ninguna explicación.

Ejemplo:

```
IF (.NOT. SW) THEN
ENTDIF=12.21+0.25_* FLUJO
ELSE
   DRVENT=(ENTSAL+ENDTIF_*ENTDER)/0.438
END IF
```

Existen equipos que permiten definir un área de parámetros e incluirla en el módulo en el momento de ejecución.

Todos aquellos parámetros deberán declararse explícitamente según las condiciones de cada compilador.

Ejemplo:

```
INTER LU, MA, MIE, JUE, VIE, SAB, DOM
DATA  LU, MA, MIE, JUE, VIE, SAB, DOM
       1,  2,   3,   4,   5,   6,  7/
```

```
C    ------------------------------------
C    * REALIZAR EL PROCESO PARA LOS SIETE*
C    * DIAS DE LA SEMANA                 *
C    ------------------------------------
   FOR      I=LU, DOM
        ---------------------------
        ---------------------------
        ---------------------------
   END FOR
```

10. Instrucciones Prohibidas.

Todo módulo (programa principal o subrutina) deberá ser escrito utilizando comandos secuenciales y las estructuras lógicas de control definidas en la sección 3.3.3 ya que el uso de otro tipo de comandos y estructuras sólo obscurece el contenido y desagrada el producto.

Es por ello que se sugiere eliminar, hasta donde sea posible el uso de las estructuras e instrucciones que a continuación se mencionan.

10.1 GO TO

Es el comando más dañino de todos. Debe ser evitado en to--

das sus ascepciones, excepto en compiladores o interpretes que no contengan instrucciones estructuradas.

10.2 Equivalencia

El uso de equivalencias se justifica solamente cuando se - desea ahorrar espacio en memoria.

La equivalencia, utiliza dos variables diferentes para representar la misma información; es muy confuso y obscurece la lógica del programa.

Para finalizar, la figura 4.2.3.2 muestra un ejemplo de una subrutina completa donde se presentan algunos de los puntos anteriores.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

GRAFICAS

FEBRERO, 1984

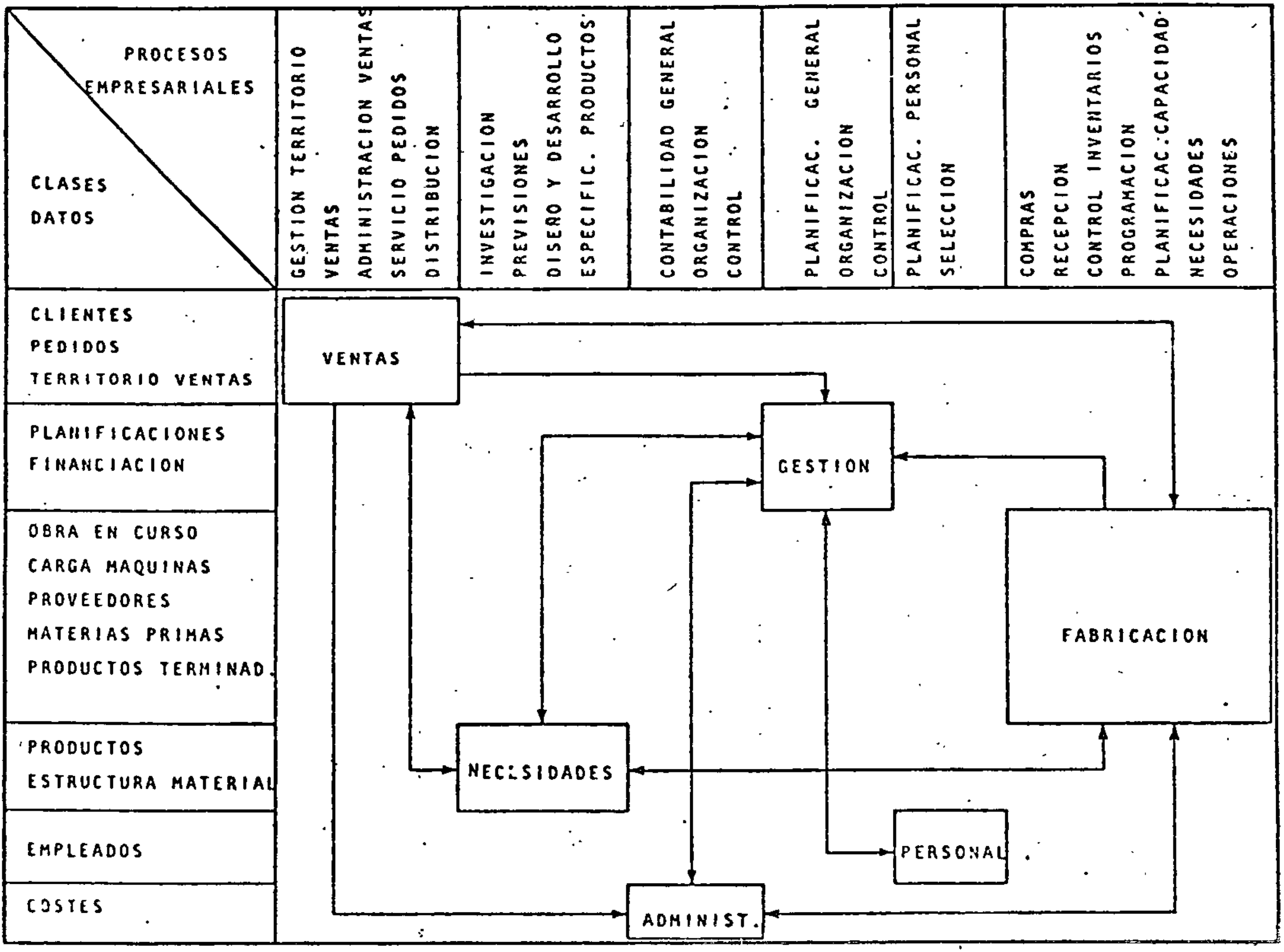
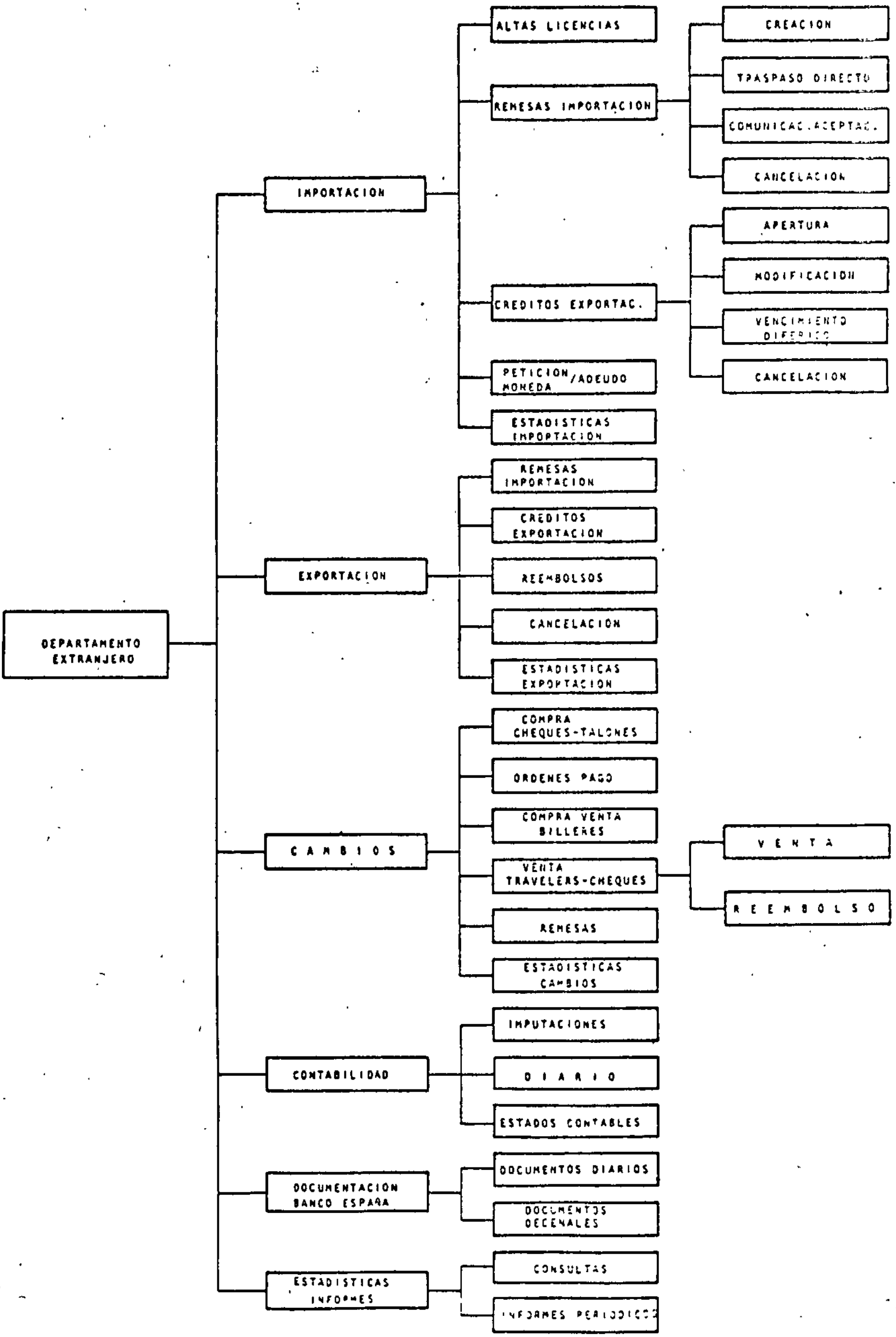


FIGURA 1. ARQUITECTURA DE INFORMACION

FIGURA 2. ESTRUCTURA SISTEMA INFORMACION



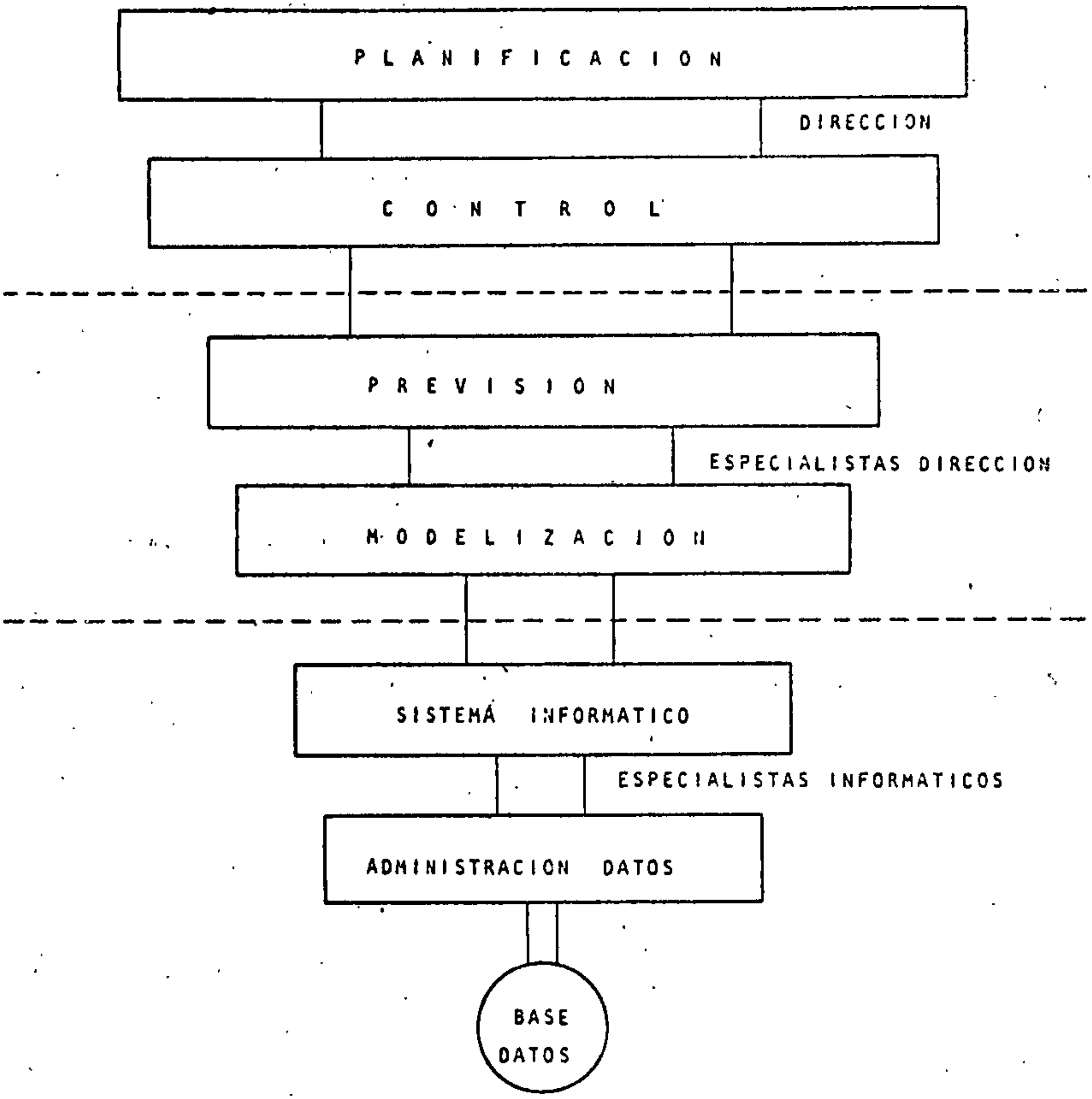


FIGURA 3. NIVELES DEL MIS

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

CONSTRUCCION DE PROGRAMAS

(Complemento)

FEBRERO, 1984

INTEGRACION

En esta sección se presentan algunos procedimientos para integrar y probar módulos de programas.

Los errores que se llegan a presentar en productos de programación pueden ser clasificados según el siguiente criterio:

- ERROR DE CODIFICACION
- ERROR DE ANALISIS
- ERROR DE ESPECIFICACION
- ERROR DE DISEÑO

Los errores de codificación son identificados durante esta etapa de integración. Los restantes tipos de error son tema de la siguiente sección denominada Pruebas de Alto NIvel.

En la descripción de la fase de arquitectura se sugirió utilizar el diseño de arriba-hacia-abajo. En la fase de construcción se expusieron dos estrategias para codificar los módulos (Top-Down y Botom-up). Un programa de computadora originalmente diseñado de arriba-hacia-abajo o de abajo-hacia arriba, puede ser codificado, integrado y probado de arriba-hacia-abajo o de abajo-hacia arriba.

TECNICAS DE INTEGRACION

A continuación presentamos dos procedimientos que pueden seguirse en la integración y validación de los módulos.

1. Integración y pruebas no incremental, donde la validación de programas de computadora se hace a partir de pruebas modulares independientes.

2. Integración y pruebas incremental, donde se realiza la validación de nuevos módulos (no probados) agregandolos a módulos ya probados e integrados.

Sobre estos procedimientos puede comentarse lo siguiente:

El proceso de integración incremental requiere un menor esfuerzo ya que por ejemplo, empieza a integrar y probar los módulos del nivel jerárquico más abajo dentro de un diagrama de estructura, es decir, con aquellos módulos "terminales" que no requieren --- llamar a otros módulos. En cambio, al utilizar la estrategia no incremental será necesario aplicar mayor esfuerzo, ya que para - probar un módulo "direccionador" o se el que llama a otros a ejecución, habrá que diseñar módulos "ciegos" que simulen las --- funciones de los subordinados, una tarea que generalmente resul ta más complicada que la generación de módulos "Direccionadores".

La función de los módulos "Ciegos" es generalmente mal interpretada, ya que no debe ser su función alarmar"llegue a la subrutina B", o simplemente regresar el control al nivel jerárquico superior, sin antes simular su funcionamiento.

Además, la integración incremental permite identificar errores - en los módulos del programa de computadora adicionados mas re--- cientemente. Integración no incremental elimina esta identificación, ya que los errores en este caso, emergirían hasta que to-- dos los módulos hubieran sido ensamblados; estos errores podrían estar en cualquier módulo del programa de computadora volviéndose muy dificil la identificación de los componentes incorrectos.

Otra ventaja asociada con la integración incremental es evitar - la posibilidad de traslapar las fases de diseño y pruebas, ya - que no debe comenzar la etapa de pruebas hasta no haber diseñado el último de los módulos del programa.

Asímismo, el problema de dejar inconclusas las pruebas de un módulo, comenzando las de otro, debido a las diferentes funciones por simular y codificar en los módulos "ciegos" desaparece en la integración incremental.

Los problemas asociados con la imposibilidad o dificulatad de - generar casos de prueba dentro de una estrategia no incremental, no existen en la incremental, ya que se asocia un caso de --- prueba a cada módulo "Direccionador". Las pruebas se imponen directamente a los módulos subordinados, evitando la necesidad de contemplar otro módulos del programa de computadora.

En conclusión, podríamos señalar que dadas las tendencias en -- los sistemas de cómputo- costos decrecientes del equipo y cre-- cientes en la programación- los errores de programación contribuyen a engrosar los costos de programación; detectarlos oportunamente ayuda a disminuir el precio de la programación. La complejidad en el diseño de casos de prueba de la estrategia no incremental no favorece precisamente esta detección oportuna de - errores , por lo cual nos orilla a respaldar la estrategia in-- cremental en la integración y pruebas de programas de computadora.

PRUEBAS DE ALTO NIVEL

Como ya se mencionó anteriormente, en el proceso de desarrollo de sistemas es usual tener una serie de errores en las primeras fases del ciclo, los cuales son detectados en las fases finales. Esto coadyuva a incrementar el costo relativo de corrección, el cual se incrementa exponencialmente.

Por otro lado, en el desarrollo de sistemas es muy usual que más del 50% de costos y tiempo sean empleados en probar el sistema.

Por las causas anteriores es de suma importancia el saber probar los sistemas.

Para llevar a cabo una adecuada estrategia para detectar los errores del sistema (probarlo) deben contemplarse métodos tanto con ayuda como sin ayuda de la computadora. El primero consiste básicamente en inspecciones y recorridas (walkthroughs) de los diferentes productos generados en el desarrollo del sistema, y el segundo consiste en diseñar casos de prueba para detectar los errores del programa cuando está ejecutándose.

El objetivo de probar.

Existen ideas errőneas del objetivo de probar, entre las que pueden considerarse:

a) "Probar es un proceso que demuestra que el programa realiza correctamente sus funciones."

b) "Probar es un proceso que demuestra que no existen errores."

Este tipo de ideas crean problemas con el verdadero objetivo de probar, ya que al tener mal definido el objetivo, los logros alcanzados son menores.

Dado que el costo de las pruebas es alto, es de suma importancia definir el concepto real de prueba:

<u>Prueba es el proceso de ejecutar un programa con el fin de encontrar errores</u>

Además no introduce calidad al programa por sí misma, sólo proporcionan una medida de control de calidad existente y puede proporcionar la extensión del error.

Como puede verse la definición presentada puede ser difícil de aceptar ya que, en lugar de ser un proceso constructivo como todo el proceso del desarrollo de sistemas, es un proceso destructivo, que su fin es encontrar todos los errores existentes.

Cuando se realiza una exitosa prueba significa que se ha detectado al menos un error, ya que si no existe detección, puede considerarse que el "programa está correcto" -que es difícil de asegurar-, o que la prueba está mal realizada.

Por otro lado cuando se realizan las pruebas con ayuda de la computadora, se necesitan diseñar casos de prueba, cuando estos casos de prueba tienen una alta probabilidad de detectar errores se dice que son "buenos", y cuando en el proceso se detectan errores, se dice que es un proceso "exitoso".

Principios de prueba.

+ El caso de prueba es un conjunto de datos de entrada y salida, donde los datos de salida son los que el programa, al estar libre de error, producirá a partir de la entrada y son el medio con el cual se prueba el programa.

  Como se observa para probar un programa debe conocerse tanto las entradas como las salidas, para que de esta manera los resultados obtenidos sean comparados correctamente y no exista el problema de "el ojo ve lo que quiere ver".

+ Un programador debe evitar probar su propio programa. Dado que el proceso de probar es netamente destructivo, es difícil que lo que se construye sea destruído por uno mismo. Este criterio es extensivo también al equipo que desarrolla sus sistemas, por lo que de preferencia deberá existir otro ente que realice el proceso para que no exista una actitud mental incorrecta.

+ Inspeccionar concienzudamente el resultado de cada prueba. Debe inspeccionarse a conciencia el resultado de cada prueba, ya que de existir errores y no detectarlos estarán desperdiciándose recursos.

+ Los casos de prueba deben ser escritos tanto para condiciones de entradas inválidas e inesperadas como para condiciones válidas y esperadas.

  La tendencia en el diseño de los casos de prueba es sólo escribirlos de acuerdo a entradas válidas y esperadas, esto es un -- error ya que el objetivo de probar es encontrar errores, por lo que los casos deben contemplar condiciones de entrada inválidas e inesperadas para que el programa sea más robusto. Con esto también se prueba qué es lo que hace el programa y qué es lo que no hace.

+ Evite los casos de prueba desechables a menos que el programa sea verdaderamente un programa desechable.
  Las pruebas que se realicen a un programa deben quedar documentadas con el objeto de no realizar nuevamente la misma prueba y poder corregir los errores con mayor facilidad.

+ No planear un esfuerzo de prueba con la suposición tácita de que no se encontrarán errores.

Cuando se realice una prueba debe tenerse la mentalidad de encontrar errores, ya que de lo contrario existirá una predisposición de no encontrarlos, con lo que se contradice el objetivo principal de probar.

+ La probabilidad de encontrar errores adicionales en una sección del programa es proporcional al número de errores ya encontrados en esa misma sección.

  La experiencia muestra que cuando en una sección se han encontrado un número n de errores, puede asegurarse que en otra sección semejante se van a encontrar igual número de errores.

+ Las pruebas constituyen una tarea altamente creativa y son un desafío intelectual.

  Para la concepción de los casos de prueba se necesita gran creatividad y un nivel de abstracción alto, ya que probar los programas extensos es más difícil que crearlos.

El proceso de probar.

Dentro de la etapa de pruebas del software se consideran básicamente 4 pasos:

a) Inspecciones y recorridas (walkthroughs) de los productos generados.

b) Diseño de los casos de prueba.

c) Ejecución de las pruebas.

d) Evaluación de las pruebas.

Como se dijo con anterioridad, el proceso de probar debe ser llevado a cabo por el hombre con o sin ayuda de la computadora.

El primer inciso se trató de una manera muy somera en el capítulo de control de calidad. En este capítulo se pretende abordarlo con mayor profundidad.

Inspecciones y Recorridas.-

El proceso de probar sin computadora (o "prueba humana") es una manera sencilla y efectiva de encontrar errores, por lo que deberán emplearse en el desarrollo de sistemas. Las inspecciones y recorridas corresponden a este método, el cual podrá aplicarse en las diferentes fases.

El objetivo de las inspecciones y recorridas no sólo es el de detectar errores de omisión, de lógica o de consistencia, sino identificar también los errores de estilo. Además, es muy importante resaltar que en estos métodos se incluyen personas con poca y mucha experiencia, transmitiéndose sus inquietudes y experiencias. Por otro lado los errores detectados no serán corregidos en la sesión.

Estos métodos se concentran básicamente en el código, aunque puede ser de diferentes tipos, los cuales se enumeran a continuación:

a) Especificación. El propósito de este tipo de método es el de indicar los problemas, inexactitudes, ambigüedades y omisiones en la especificación de requerimientos. La inspección o recorrida puede estar constituída por el usuario, el analista y uno o más diseñadores del proyecto, para examinar los diagra--

mas de flujo de datos y los demás documentos generados en el análisis estructurado.

b) Arquitectura. El propósito de esta inspección es la de indicar los defectos, inconsistencias, errores y omisiones en la arquitectura del diseño -antes de escribir el código. Estará constituída por el usuario, el analista, el diseñador y un programador. Los documentos a examinar son el diagrama de estructura, el diseño de la base y los demás documentos generados en esta fase.)

c) Pruebas. El objetivo es el de asegurar que los casos de prueba para el sistema sean los adècuados. Las personas que participan son el probador, el analista, el diseñador y el programador. En algunos casos la presencia del usuario es de suma importancia como es en la prueba de aceptación del sistema.

Las inspecciones y recorridas deberán ser realizadas con frecuencia para que pequeñas partes del producto sea revisado. Deberán ser programadas de tal manera que el programador, diseñador o analista esté listo para ello. En el caso de que los productos generados en la definición de requerimientos o en la arquitectura del sistema hayan sido terminados, se procederá a realizar una inspección.

Pero en el caso del código, las inspecciones y/o recorridas se podrán realizar en las siguientes etapas:

+ Antes de ser el código tecleado.
+ Después de ser el código tecleado, pero antes de ser compilado.
+ Después de la primera compilación.
+ Después de la primera compilación exitosa.

+ Después de ejecutar todos los casos de prueba.

Dependiendo básicamente de la complejidad del código y de su disponibilidad.

Inspección del código.

La inspección del código es realizada por un grupo de aproximadamente cuatro personas. Una de ellas es un programador de amplia experiencia, que tendrá el papel de moderador y que no es el autor del código a realizar.

Sus funciones son:

+ Distribuir el material necesario para las sesiones de inspección,
+ proporcionar el apoyo logístico de la sesión,
+ coordinar los horarios,
+ dirigir las sesiones,
+ escribir los errores detectados, y
+ verificar que sean corregidos posteriormente

Otro miembro del grupo será el programador que realizó el código y un especialista en pruebas.

Las personas que integren el grupo deben tener una actitud positiva, de tal manera que el programador no sienta que se le está corrigiendo a él, sino al programa, para que no adopte una postura defensiva, lo cual trae como consecuencia que el proceso no sea efectivo.

Una vez que el grupo está reunido, el programador que desarrolló el código empieza a realizar una presentación de su código, mientras que el resto del grupo empieza a formularle preguntas sobre el código en aquellas partes donde pueden existir errores o confusiones.

Además, el código es analizado con la ayuda de una lista de errores que comunmente existen en el código.

El programador con experiencia va escribiendo los errores detectados y al final de la sesión se los proporciona al programador. Cuando finaliza una sesión se realiza una evaluación que trae como consecuencia que el producto sea aprobado, que al producto se le hagan las correcciones pertinentes, o en el caso de existir correcciones sustanciales se propone una nueva inspección. Con los errores detectados el moderador procede a actualizar la lista que contiene errores.

La duración de estos métodos puede variar entre 60 y 120 minutos dependiendo de la complejidad y tamaño del código. Dado que el esfuerzo de cada sesión es bastante, más de dos horas puede fatigar al grupo y en consecuencia hacer improductiva la sesión.

La lista de errores es una parte medular en las inspecciones; una lista de éstos es propuesta a continuación con base a la literatura y a las experiencias.

Errores en la referencia de datos:

¿Se inicializaron todas las variables?

¿Los índices están dentro de los límites?

¿Los índices son enteros?

¿Hay atributos correctos cuando se da un cambio de nombre a un identificador?

Errores en la declaración de datos.

¿Están declaradas todas las variables?

¿Son entendidos los atributos por omisión?

¿Están correctamente asignadas las longitudes, los tipos y las clases de almacenamiento?

¿Es consistente la inicialización con el tipo de almacenamiento?

¿Hay variables con nombres similares?

Errores de cálculos.

¿Hay cálculos sobre variables no aritméticas?

¿Hay cálculos entre enteros y punto flotante?

¿Es la variable receptora de menor longitud que el tamaño asignado?

¿Hay resultados intermedios por overflow u onderflow?

¿Hay división entre cero?

¿Todas las variables toman valores dentro de su rango significativo?

¿Está entendida la procedencia de los operadores?

¿Las divisiones entre enteros están correctas?

Errores de comparación.

¿La comparación es entre variables inconsistentes?

¿Hay comparaciones mezcladas?

¿Son correctas las comparaciones de relación?

¿Son correctas las expresiones booleanas?

¿Está bien la mezcla de booleanas y de relación?

¿Está entendida la precedencia de operadores?

¿Se entiende la evaluación de las expresiones booleanas realizadas por el compilador?

Errores de flujo de control.

¿Están excedidas las ramificaciones múltiples?

¿Termina el programa?

¿Existen partes del programa que no son ejecutadas por causa de condiciones de entrada?

¿Hay errores de iteraciones en uno?

¿Se corresponden los enunciados de principio y fin de bloques de instrucciones?

Errores de interfaz.

¿El número de parámetros de entrada es igual al número de argumentos?

¿Son del mismo tipo los parámetros y argumentos?

¿Son iguales las unidades de los parámetros y de los argumentos?

¿Es correcto el orden de los parámetros y argumentos?

¿Hay constantes tomadas como argumentos?

Errores de entrada/salida.

¿Son correctos los atributos de los archivos?

¿Son correctos los comandos OPEN?

¿Las especificaciones de formato corresponden con los enunciados de E/S?

¿Es el tamaño de un área de E/S en almacenamiento igual al tamaño del registro del archivo?

¿Están abiertos todos los archivos antes de su uso?

¿Están manejadas las condiciones de cierre de archivos?

¿Están manejados los errores de E/S?

Recorridas del código.

Las inspecciones y recorridas son procesos muy similares, con la diferencia principal de que la recorrida (también llamada "Plática estructurada") es un proceso dinámico, al contrario de la inspección, que es un proceso estático de lectura, ya que se propo-- nen casos de prueba que van a ser procesados por el grupo reunido haciendo el papel de la computadora en el caso de la recorrida.

La recorrida es también realizada por un grupo de personas y tarda al igual que la inspección entre 1 y 2 horas. El grupo está integrado de 3 a 6 personas con las siguientes características.

+ un probador de gran experiencia que haga las funciones.de la misma manera oue en la inspección.

+ un experto en el lenguaje de programación en el cual el código está escrito.

+ un programador sin experiencia

+ un programador que le dará mantenimiento al código.

+ un programador que pertenezca a otro proyecto, y

+ el programador que realizó el código.

En el grupo alguien deberá hacer la función de secretario y otro la de probador.

Como se dijo, el procedimiento es similar al de inspección con la diferencia de que en lugar de leer el programa y ayudarse de la lista de errores, los integrantes "realizan la función de la computadora", procesando unos casos de prueba pequeños y representativos previamente definidos que den lugar a cuestionar el programa. El estado del programa es plasmado en papel o en el pizarrón para ser analizado. Cuando finaliza la sesión deben evaluarse los resultados para que el código sea aprobado, se le hagan las correcciones pertinentes o se realice una nueva recorrida después de haber corregido los errores.

Otra práctica, ya muy difundida, es la prueba de escritorio, la cual es semejante a las inspecciones y recorridas, pero con una sola persona.

Diseño dé los casos de Prueba.-

Para probar un programa se necesita:

+ Ejecutar el programa con un conjunto de datos de entrada.

+ Observar el efecto de las entradas en la ejecución.

+ Examinar las salidas del programa por medio de salidas esperadas para determinar su aceptación.

Estos conjuntos de datos deberán ser diseñados para realizar la prueba, de tal manera que tengan la mayor probabilidad de detectar el mayor número posible de errores. Una manera sencilla es que los datos sean generados al azar, teniéndose algunas dificultades y carencias, especialmente si sólo se realiza esta clase de prueba. Para una mejor opción existen dos técnicas principales para llevar a cabo las pruebas: la de caja negra y la de caja transparente.

La premisa fundamental de caja transparente es que la prueba puede ser diseñada de acuerdo a la lógica interna del programa.

La premisa fundamental de caja negra es que la prueba puede ser diseñada de acuerdo a la función que ejecuta, esto es, sólo se analizan las entradas y las salidas.

Se recomienda que ambas técnicas sean utilizadas para llevar a cabo una prueba más rigurosa, ya que cada una tiene sus pros y contras.

Técnica de Caja Transparente.-

Las pruebas de caja transparente (caja blanca) o lógicas, permiten diseñar los casos de prueba de acuerdo a la estructura interna del programa. Una idea errónea es querer obtener los casos de prueba de tal manera que todas las secuencias lógicas del programa sean ejecutadas, dando lugar a realizar una prueba exhaustiva, ya que en la mayoría de los casos es imposible por el tiempo requerido para realizarlo; por lo que es recomendable seleccionar un limitado conjunto de secuencias lógicas "importantes".

Por otro lado, el ejecutar todas las secuencias lógicas no garantiza que el módulo realice bien su función, ya que la lógica del programa puede estar correcta pero no es el resultado deseado, o pueden faltar secuencias, lo cual no daría tampoco el resultado deseado.

Los criterios para el diseño de casos de prueba con la técnica de caja blanca se describen a continuación:

Cobertura de instrucciones. La idea de este criterio es que todas las instrucciones del programa sean ejecutadas al menos una vez. Este criterio es muy deficiente por lo que se considera inútil.

Cobertura de decisiones. Este criterio establece que cada secuencia lógica de instrucciones (lazo) debe ejecutarse una vez sí y otra no; esto puede traducirse a que cada decisión debe tener un falso y un verdadero.

Cobertura de condiciones. Este criterio establece que cada condición en cada decisión tenga todos los resultados posibles. Esto es, que si una decisión contiene varias condiciones, éstas deberán tener todos los resultados posibles.

Cobertura de condición múltiple. Este criterio establece que todas las combinaciones posibles de resultados de condición en cada decisión y todos los puntos de entrada se invoquen por lo menos una vez.

Técnica de Caja Negra.-

Las pruebas de caja negra o producidas por entrada/salida permiten diseñar los casos de prueba en función de las especificaciones del programa sin tener necesidad de consultar su lógica interna. Se tiene la ventaja, por lo tanto, de que cuando se diseñan los casos de prueba no es necesario conocer el código, pudiendo además desarrollar el código en forma paralela. Por otro lado al momento de revisar los requerimientos se pueden detectar errores.

Los criterios para el diseño de casos de prueba con esta técnica se presentan a continuación:

Particiones de equivalencia. Dado que la prueba exhaustiva de entrada a un programa es prácticamente imposible, puede pensarse en definir un caso de prueba que cubra una clase de errores, consiguiendo con esto reducir el número total de casos de prueba. Para definir esta clase de errores debe tomarse en cuenta: a) que reduzcan segnificativamente el número de los otros casos de prueba que deben diseñarse para alcanzar una meta predefinida de prueba "razonable"; y b) que cubran un conjunto extenso de otros casos de prueba posibles.

Con esto, puede decirse que con un caso de prueba se cubrirá el número mayor de entradas diferentes con el objeto de reducir los casos de prueba. Por otro lado al proponer un caso de prueba, se omitirán todos los demás que pertenezcan a la misma clase, ya que detectarán el mismo error.

Tomando en cuenta las dos consideraciones anteriores, se forma el criterio de partición de equivalencia, ya que se realiza una partición del conjunto de casos de pruebas que pueden ser cubiertas con el mínimo número de casos de prueba.

Este criterio está compuesto de dos pasos:

a) identificación de las clases de equivalencia, y

b) definición de los casos de prueba.

La identificación de las clases de equivalencia se hace dividiendo las condiciones de entrada (usualmente una especificación) en subconjuntos de válidas e inválidas. Por ejemplo, una condición válida para el RFC es:

Iniciales	4 letras
Año	2 dígitos
Mes	2 dígitos entre 1 y 12
Día	2 dígitos entre 1 y 31

una condición inválida para el ejemplo anterior sería: mes> 12, o iniciales AA9J.

Para poder manejar con mayor claridad este concepto se recomienda utilizar la siguiente tabla:

Condición externa	Clases de equivalencia válidas	Clases de equivalencia inválidas

El segundo paso, el de definición de los casos de prueba, consiste en

a) Asignar un número único a cada clase de equivalencia.

b) Escribir un caso de prueba que cubra el máximo de equivalencias válidas que no han sido cubiertas, repitiendo el proceso hasta que no existan equivalencias válidas.

c) Escribir un caso de prueba para cada una de las equivalencias inválidas.

El objeto de cubrir cada una de las equivalencias inválidas es que en algunas ocasiones, cuando se detecta una, no pueden detectarse más.

Análisis de valores límite. Dado que el criterio de particiones de equivalencia selecciona las clases, el criterio de análisis de valores límite puede hacer uso de cada una de estas clases, de tal manera que el límite de cada clase sea cubierto, además de enfocar la atención en los límites de las entradas y de las salidas.

En el ejemplo anterior, mes es válido entre 1 y 12, por lo que aplicando este criterio, unos casos de prueba son 0, 1, 12 y 13 para el mes. Este criterio se aplica tanto a rangos (1-12) como a número de valores (20 registros) tanto para entradas como para salidas.

Otro ejemplo es cuando se quieren imprimir encabezados y 50 nombres en cada página. Un caso de prueba es cuando se tienen 50 nombres y otro cuando se tienen 51 nombres; con esto se detecta si la impresión de encabezados se realiza correctamente.

Conjetura de errores. En la utilización de este criterio se pone en práctica la intuición y la experiencia para detectar errores.

Con la ayuda de una lista de errores (semejante o igual a la de inspecciones) pueden diseñarse casos de prueba que no podrían ser cubiertos con las técnicas anteriores.

Gráficas causa-efecto. Este criterio se basa en la premisa de que el flujo lógico del programa puede graficarse. Las gráficas de causa-efecto son un criterio para probar que proporciona una representación concisa de las condiciones lógicas y sus acciones correspondientes, el criterio tiene cuatro pasos:

a) las causas (condiciones de entrada) y los efectos (acciones) de un módulo son listados, y un identificador es asignado a cada uno.

b) Una gráfica causa-efecto es desarrollada.

c) La gráfica se convierte a una tabla de decisión.

d) Las reglas de la tabla de decisión son convertidas a casos de prueba.

Ejecución de las pruebas.-

Una vez que se han diseñado los casos de prueba, se procede a ejecutarlos para de esta manera detectar los errores.

El diseño de los casos de prueba se tiene que realizar de acuerdo al nivel y tipo de prueba que se requiera.

Existen básicamente 5 tipos de prueba que están divididos en dos niveles:

Nivel	Tipo
Pruebas de Bajo Nivel	Modulares o de construcción integración
Pruebas de Alto Nivel	Funcionales Sistema Aceptación

La razón de efectuar pruebas de diferentes tipos es la de detectar errores que puedan ser atribuidos a las diferentes fases del desarrollo del sistema. Por ejemplo, si existen errores en la prueba de integración, puede atribuirsele a un mal diseño, ya que pueden tenerse problemas de interfase.

Pruebas modulares. Es conveniente como en todo proceso, empezar desde lo más pequeño y menos complejo; siguiendo esta idea, en el software la parte más pequeña considerada como unidad es un módulo, una subrutina o un procedimiento, por lo que se procede a primera instancia a probar los módulos.

Otras ventajas de empezar con este tipo de prueba, es que los errores son detectados y corregidos con mayor facilidad, además de poder probar módulos en paralelo.

El objetivo de este tipo de prueba es el de detectar errores en la codificación.

La técnica utilizada para el diseño de los casos de prueba es la de caja transparente, ya que la lógica interna de un módulo no es tan difícil de seguir. Por otro lado cuando se diseñan los casos de prueba con la otra técnica, puede facilitarse si el módulo tiene un alto grado de cohesión, traduciéndose esto a que el módulo desarrolle sólo una función y por ende el número de casos de prueba sea reducido. El problema fundamental de este tipo de prueba radica en la simulación de módulos vecinos. Una ayuda extra para el diseño de los casos de prueba de este tipo es la lista de errores de las inspecciones.

Pruebas de Integración.-
Las pruebas de integración como su nombre lo dice, se realizan sobre algo integrado, en este caso los módulos.

Una vez que los módulos han sido probados individualmente y funcionan correctamente, puede pensarse en que al momento de juntarlos (integrarlos) unos con otros van a funcionar correctamente, pero por desgracia esto no es verídico, ya que pueden tenerse problemas en las interfases, o la función que realiza un determinado módulo afecta a otro. Es por esta razón que se realiza la prueba de integración.

El objetivo de esta prueba es localizar los errores generados en la codificación y en el diseño.

Para realizar las pruebas de integración se recomienda utilizar la técnica de caja transparente cuando la lógica no sea compleja.

Se tienen dos formas de realizar la integración:

a) Cuando todos los módulos han sido probados y luego juntarlos, o

b) integrando cada nuevo módulo a probar con los demás probados previamente antes de probar el programa.

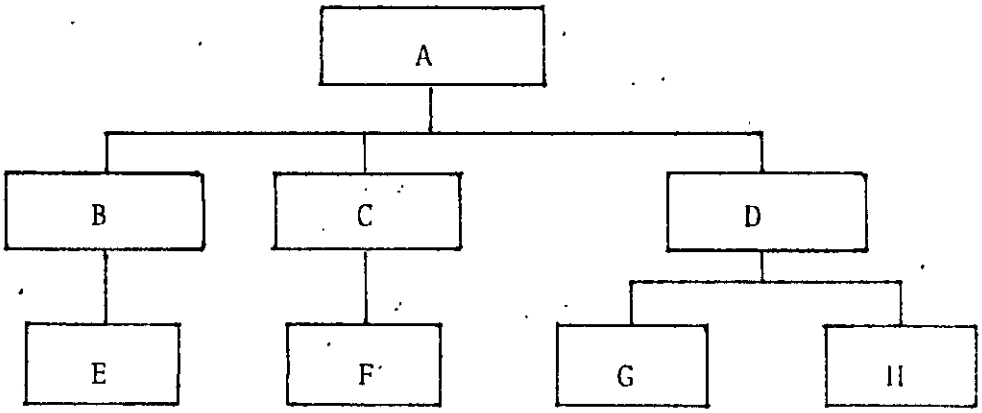
Como se observa en el primer inciso se trata de la realización de la prueba modular y luego la integración de todos para su prueba, a esta forma se le conoce como prueba no incremental o de gran explosión.

En el segundo caso, la prueba modular no se ejecuta completamente, sino que cada nuevo módulo a probar se integra a los ya probados, a esta forma se le denomina prueba incremental.

Dado que dependiendo de la forma en que se realicen las pruebas de integración, se puede considerar que las pruebas modulares pueden o no existir y ser parte de las de integración.

Para aclarar el concepto de pruebas incrementales y no incrementales, se presenta el siguiente ejemplo:

Dada la siguiente estructura:



La manera de realizar las pruebas no incrementales es probar cada uno de los módulos para posteriormente ser integrados y probados. En el momento de probarse el módulo D, se requiere un módulo impulsor especial (que haga la función de A y dos módulos auxiliares que hagan la función de G y H).

La creación de los módulos impulsores no es un proceso sencillo, ya que la interfase debe estar bien definida para saber qué datos mandará. Por otro lado, la creación de los módulos auxiliares es más compleja, ya que éstos como se piensa no solo reciben información, sino que también transmiten y en algunas ocasiones son invocadas iterativamente. La creación de este tipo de módulos crea una situación desventajosa.

Una ventaja importante es que las pruebas de los módulos pueden realizarse en paralelo.

En la realización de las pruebas incrementales existen dos métodos: el ascendente y el descendente. Tomando como referencia la figura anterior y el método descendente, el primer módulo que se prueba es el A y luego se continúa con cualquiera en el cual su módulo superior haya sido invocado (p.e. una secuencia sería ABECDFGH). Si el método es ascendente se prueban todos los módulos de nivel inferior, para continuar con cualquiera que tenga todos sus subordinados probados (p.e. una secuencia sería EFGHBCDA).

Cuando se utilice la forma incremental se recomienda que los módulos que se escojan sean de entrada y/o salida, ya que de esta manera se tendrán menos dificultades en la creación de los módulos impulsores y auxiliares, o que los módulos tengan una lógica compleja para una rápida corrección de sus errores.

Se puede considerar que se tienen ciertas ventajas con la forma incremental ya que los errores de interfase son detectados más rápido, si se utiliza el método ascendente no se necesitan crear módulos auxiliares y las correcciones serán más rápidas.

Pruebas funcionales. Las pruebas funcionales son el primer tipo de pruebas de alto nivel. Las pruebas de alto nivel deben ser realizadas a programas que tienen un cierto grado de complejidad, calidad y documentación como son los programas producto (sistemas). Este tipo de programas contienen requerimientos y objetivos formales que pueden ser comparados con las pruebas de alto nivel. Pero en aquellos programas de baja complejidad la prueba de alto nivel consistirá sólo de la prueba funcional.

Las pruebas funcionales tienen como objetivo principal encontrar errores de especificación de requerimientos. Estos errores son creados por las discrepancias existentes entre el análisis y la especificación.

Para el diseño de casos de prueba es recomendable utilizar la técnica de caja negra, ya que la técnica de caja transparente es irrelevante por la necesidad de verificar los requerimientos. En la selección de los casos de prueba es importante basarse en la especificación de requerimientos.

Dado que un error de software es aquel que se presenta cuando el programa no hace lo que el usuario espera razonablemente que haga, las pruebas de alto nivel deberán ser reportadas formalmente en un documento denominado plan de pruebas, para ser en algunos casos aprobado por el usuario.

Se recomienda que este tipo de pruebas sean llevadas a cabo por una persona con capacidad de abstracción para una mejor selección de los casos de prueba.

Pruebas de sistema. La prueba del sistema es el tipo de prueba más difícil, ya que no es un proceso para verificar que las funciones del sistema sean correctas -propósito de la prueba funcional- sino que su objetivo es el identificar las discrepancias entre el producto original y los objetivos originales del sistema.

Para diseñar los casos de prueba se utiliza la técnica de caja negra, tomando como fuente principal de información el manual de usuario, las documentaciones destinadas a él y los objetivos del sistema. Cuando se tienen errores en la documentación se procede a corregirla.

En este tipo de pruebas no se tiene una metodología clara, ya que dependen básicamente del tipo y complejidad del sistema. Se pueden tener entre otras, las siguientes categorías para el diseño de los casos de prueba:

+ Pruebas de volumen
+ Pruebas de esfuerzo
+ Pruebas de facilidad de uso
+ Pruebas de seguridad
+ Pruebas de compatibilidad y conversión
+ Pruebas de facilidad de instalación
+ Pruebas de confiabilidad
+ Pruebas de documentación

Es importante recalcar que este tipo de pruebas son difíciles, por lo que se necesitará gran creatividad, ingenio y experiencia.

Prueba de Aceptación. El objetivo básico de las pruebas de aceptación es comparar el producto final con el contrato original. Esta prueba será conducida por el usuario final, o en algunos casos dependiendo de la complejidad, se realiza por un tercer ente. Estas pruebas pueden ir desde una prueba similar a las anteriores hasta una sistemática y planeada ejecución de casos de prueba. En función de los resultados y evaluación de la prueba el producto será o no aceptado por el usuario.

En la siguiente tabla se presentan las características principales de los tipos de pruebas expuestos.

Tipo de Prueba	Tipo de error que identifica	Técnica recomendada
Modular	Código	Caja transparente
Integración	Código y diseño	Caja transparente
Funcional	Especificación	Caja negra
Sistema	Objetivos,especificación y diseño	Caja negra
Aceptación	Especificación y diseño	Caja negra

La ejecución de las pruebas y su diseño debe ser planeada y controlada,.dado que no es un proceso sencillo, sino al contrario es un proceso de gran envergadura que utilizará un gran despliegue de recursos.

Para la exitosa realización de las pruebas debe existir una planeación que deberá contener los siguientes aspectos:

a) Objetivos. Deberán definirse los objetivos de cada tipo de prueba.

b) Criterios de terminación. Deberá definirse bajo qué criterios termina cada tipo de prueba.

c) Calendario. Calendarizar la ejecución de los diferentes tipos de prueba, tomando en cuenta cuándo se diseñarán, escribirán y ejecutarán.

d) Responsabilidades. Asignar responsabilidades de las diferentes actividades en el proceso de desarrollo de pruebas (p.e. quién diseñará, quién escribirá y quién ejecutará).

e) Herramientas. Definir qué herramientas y apoyos se utilizarán en el proceso de las pruebas (p.e. bibliotecas de código reusable, generadores de archivos de prueba).

f) Recursos de máquina. Hacer una estimación.del tiempo y recursos necesarios de máquina.

g) Integración. Qué tipo de forma de integración se utilizará (p.e. incremental).

h) Métodos de seguimiento. Identificar los medios para estimar el avance del proceso de plan de pruebas.

i) Procedimientos de corrección. Definir el mecanismo por el cual se reportarán los errores y se realizarán las correcciones al sistema.

j) Pruebas de deterioro. Ejecución de pruebas de las partes del sistema que puedan ser alteradas por la corrección de un error.

Para la ejecución de cada prueba, se contará previamente con los casos de prueba, la descripción de estos casos se denominará plan de pruebas. Además se tendrán que considerar los apoyos logísticos (p.e. personal, recursos de cómputo e interfaz), el documento que describa estos apoyos se le llamará procedimientos de prueba.

Evaluación de las Pruebas.-

Una vez que se han realizado las pruebas se realizará una evaluación para proceder a efectuar las correcciones de una manera sistemática y organizada, contemplando la manera en que se realizarán las modificaciones, incluyendo calendarios, responsables, herramientas y tiempos. Además deberá verificarse qué otros módulos serán afectados con el cambio para tomar las medidas adecuadas.

Normas para la documentación de planes de prueba.-

Las normas para la documentación de planes de prueba tienen como objetivo tener uniformidad de criterios en la elaboración de los planes de prueba, sin limitar la creatividad del responsable y colaboradores, además de lograr homogeneidad e integridad en la documentación.

El documento de planes de prueba estará organizado de la siguiente manera:

- Documento de Planes de Prueba
  - Portada
  - Introducción
  - Capítulos (n)
    - Portada
    - Contenido
    - Secciones

Por cada sistema o proyecto se contará con un documento de plan de pruebas. Dicho documento estará dividido en una portada y varios capítulos donde el primero será la introducción y los restan tes los subsistemas o subproyectos, identificados por sus respectivos nombres.

La portada del documento de plan de pruebas estará compuesta de dos hojas conteniendo la siguiente información:

- portada del plan de pruebas
  - primera hoja
    - nombre de la institución
    - nombre del documento
    - nombre del departamento que elaboró
    - nombre del departamento que revisó
    - fecha de la última revisión
    - número de la hoja
  - segunda hoja
    - nombre de los participantes
    - número de la hoja

La portada del documento de plan de pruebas se ejemplifica con las figuras 1 y 2.

La introducción del documento de plan de pruebas contendrá el objetivo del plan de pruebas y un panorama general del contenido del plan.

El resto de los capítulos será un número igual al número de subsistemas o subproyectos del sistema o proyecto.

Cada capítulo recibirá el nombre del subsistema o subproyecto a excepción de la introducción, y serán numerados en orden progresivo.

A excepción de la introducción, cada capítulo contendrá una portada, una lista de su contenido y un número variable de secciones.

La portada del capítulo contendrá la siguiente información:

portada del capítulo
- nombre de la institución
- número y nombre del capítulo
- nombre del responsable
- nombre de los colaboradores
- nombre de los que autorizan
- fecha de la última revisión
- número de la página
  - número del capítulo
  - guión
  - número consecutivo inicial del capítulo

La portada del capítulo se ejemplifica con la figura 3.

La lista del contenido del capítulo mostrará las secciones de las que está compuesto el capítulo. Será presentada después de la portada del capítulo.

Cada sección será identificada por un número compuesto del número del capítulo y el número consecutivo de la sección, separadas por un punto, organizándolas de la siguiente manera:

secciones
- introducción
- referencias
- pruebas propuestas
  (m)

La introducción de las secciones estará dividido en tres partes:

Párrafo inicial. Identificará el nombre del proyecto o subproyecto al cual corresponde el plan de pruebas.

Panorama general del capítulo. Describe un panorama general de la estructura y organización del capítulo, conteniendo el número total de secciones además de las secciones que pueden ser realizadas en paralelo y/o en serie.

Resumen del plan de pruebas. Se presentará en forma tabular la información siguiente:

+ Clave de la prueba (compuesta por el número del capítulo y el número consecutivo de la prueba del capítulo separado por un punto),

+ descripción breve, y

+ número de casos de prueba.

El resumen queda ejemplificado con la figura 4.

La lista de referencias deberá contener las referencias bibliográficas utilizadas en el capítulo. Estas referencias incluyen los fundamentos de análisis para la selección de entradas y/o salidas, las normas a seguir para documentar los planes de prueba y las técnicas para diseñar los casos de prueba.

Cada referencia tendrá un número consecutivo por capítulo.

Cada sección tendrá un número variable de pruebas propuestas, a las que se les asociará una clave (compuesta por el número del capítulo y el número consecutivo de la prueba del capítulo, separados por un punto). Cada prueba propuesta deberá contener la siguiente información:

pruebas propuestas {
portada
análisis
identificación de entradas y salidas
reseña
casos de prueba

Esta información es la que servirá como documentación principal, ya que en base a ella, personas no involucradas en su elaboración podrán revisar y evaluar el diseño de los casos de prueba.

La portada de una prueba propuesta contendrá la siguiente información:

portada de una prueba propuesta

- número y nombre del capítulo
- clave de la sección
- clave y nombre de la prueba
- nombre de la persona que elaboró
- fecha de la última revisión
- número de la página
  - número del capítulo
  - guión
  - número consecutivo del capítulo

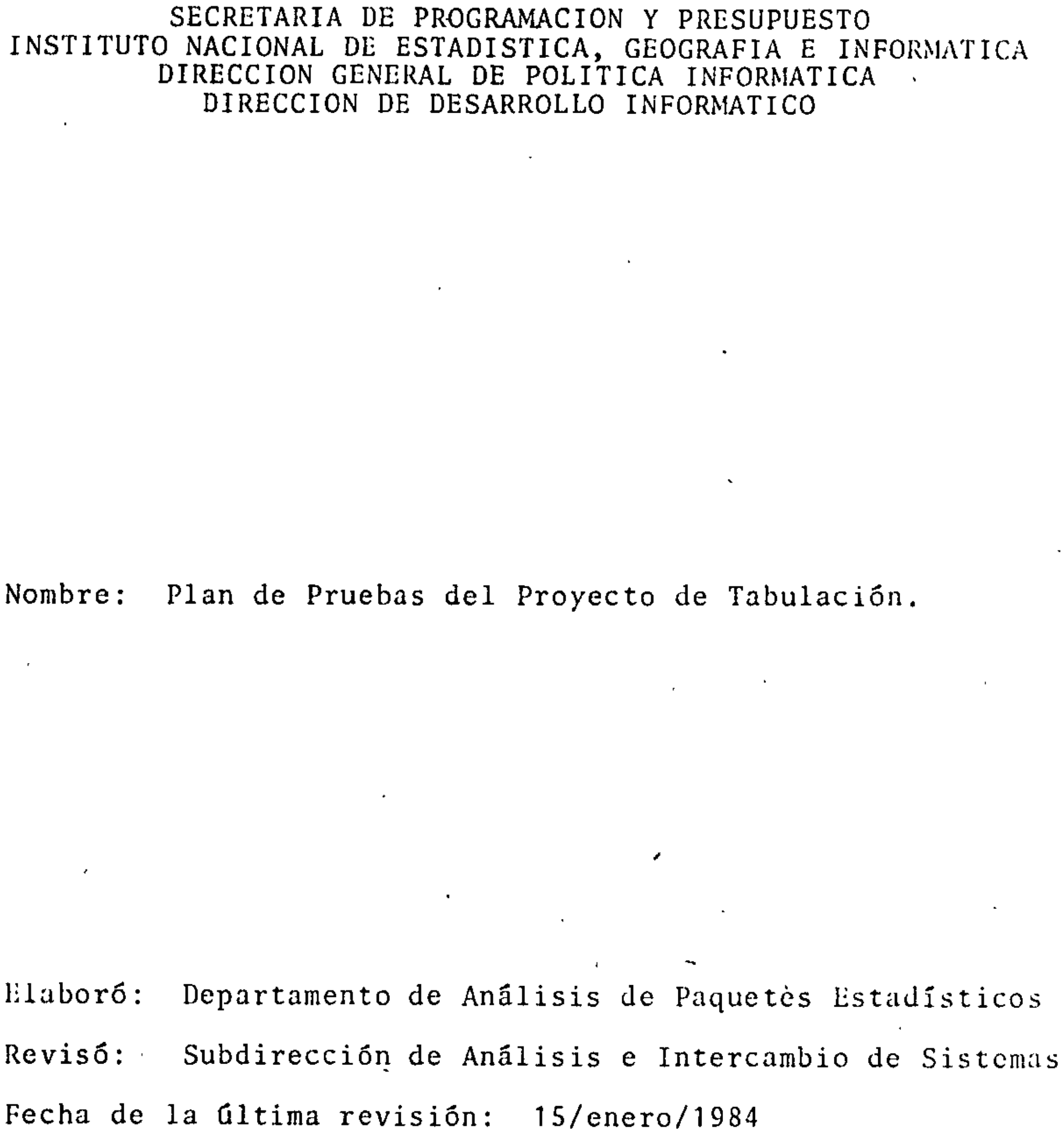
La portada de una prueba propuesta se ejemplifica en la figura 5.

El análisis describirá cuál es el requerimiento u objetivo que va a probarse, cuál es el concepto clave sobre el cual se fundamenta la solución del requerimiento y además cuáles son las características susceptibles a tener errores.

La identificación de entradas y salidas, consistirá en la documentación en forma tabular de las condiciones de entrada y salida, agrupándolas en clases válidas e inválidas de equivalencia con sus respectivos identificadores. La tabla se ejemplifica en la figura 6.

La reseña de la prueba contendrá el número total de los casos de prueba, el criterio o criterios utilizados para seleccionar los casos de prueba, el método o métodos que serán utilizados para verificar los resultados de la prueba (p.e. inspección de listado, inspección de desplegado) y el criterio de aceptación de resultados de la prueba. Además, cuando la prueba se efectúe a un sistema que tenga como entrada, además de los datos a procesar, un pro grama propio en su propio lenguaje (preprocesador) o en un lenguaje huésped (p.e. FORTRAN) se deberá incluir dicho programa con su respectiva documentación.

Se presentarán todos los casos de la prueba, propuestos en la última subsección de las pruebas propuestas (los casos de la prue--ba) de manera tabular de preferencia. Los casos de prueba identificados por una clave compuesta por tres números separados por puntos (los dos primeros corresponderán a la clave de la prueba y el tercero será el identificador de la clase), deberán estar compuestos por un conjunto de datos de entrada asociados con sus correspondientes datos de salida.



SECRETARIA DE PROGRAMACION Y PRESUPUESTO
INSTITUTO NACIONAL DE ESTADISTICA, GEOGRAFIA E INFORMATICA
DIRECCION GENERAL DE POLITICA INFORMATICA
DIRECCION DE DESARROLLO INFORMATICO

Nombre: Plan de Pruebas del Proyecto de Tabulación.

Elaboró: Departamento de Análisis de Paquetès Estadísticos

Revisó: Subdirección de Análisis e Intercambio de Sistemas

Fecha de la última revisión: 15/enero/1984

FIG. 1

Primera hoja de la portada del plan de pruebas

Participantes: Inés González
José Flores
Benito Zychlinski

FIG. 2

Segunda hoja de la portada de plan de pruebas

SECRETARIA DE PROGRAMACION Y PRESUPUESTO
INSTITUTO NACIONAL DE ESTADISTICA, GEOGRAFIA E INFORMATICA
DIRECCION GENERAL DE POLITICA INFORMATICA
DIRECCION DE DESARROLLO INFORMATICO

Capítulo: 2

Nombre: Plan de Pruebas del Paquete SPSS en el Proyecto de Tabulación

Responsable: Inés González

Colaboradores:

Autorizó: José Flores
Benito Zychlinski

Fecha de la última revisión: 10/enero/1984

2-1

FIG. 3

Portada del Capítulo

Clave de la Prueba	Descripción	Número de Casos
2.1	Prueba de volumen	10

FIG. 4
Resumen del plan de pruebas

Capítulo: 2

Nombre: Plan de Pruebas del Paquete SPSS en el proyecto de Tabulación

Sección: 2.3

Prueba: 2.1

Nombre: Prueba de volumen

Elaboró: Inés González

Fecha de la última revisión: 10/enero/1984

2-8

FIG. 5

Portada de prueba propuesta

Condición Externa	Clase Válida de Equivalencia	Clase Inválida de Equivalencia
Entradas		
número de registros	0- 10,000 (1)	10,000 (2)
número de tabulados	1 - 15 (3)	0 (4)
Salidas		
mensaje "exceso de registros'	10,000 (2)	15 (5)
cambio de hoja por tabulado	2 - 15 (4)	0 - 10,000 (1)

FIG. 6

Tabla de identificación de entradas y salidas

Herramientas Automáticas.-

El proceso de probar es bastante costoso y tardado, por lo que se han realizado bastantes investigaciones sobre herramientas automáticas para abatir estos problemas.

Las herramientas automáticas pueden dividirse en dos grandes rubros para el apoyo en el proceso de probar:

+ Estáticas. Que apoyan al proceso de probar sin necesidad de ejecutar el programa.

+ Dinámicas. Que apoyan al proceso de probar en la ejecución del programa o una vez ejecutado.

Dentro de las herramientas automáticas se tiene:

Analizador estático de código. Descrito previamente (capítulo 3). Sirve básicamente para determinar errores de inicialización y violaciones a las normas de codificación previamente establecidas.

Biblioteca de código reusable. Descrito previamente (capítulo 3). Sirve básicamente para proporcionar casos de prueba previamente diseñados de una manera organizada y automatizada.

Recopilador de errores. Es un programa que da de alta los errores comunes que existen dentro del software. Como se sabe, el tener un conjunto de errores y sus causas en el proceso del desarrollo del software no es cosa fácil, por lo que al tener un recopilador de este tipo de información es de gran utilidad para el diseño de los casos de prueba o para el desarrollo de los futuros sistemas -es tener una lista de errores automatizada. Por otro lado, para hacer más valiosa esta herramienta, a cada error y su causa puede adicionársele los métodos más efectivos para prevenirlos o detectarlos previamente, así como la manera de corregirlos.

En la actualidad existen estudios que incluyen el análisis de los errores encontrados durante la prueba de una emisión del sistema operativo DOS/VS de IBM*; un estudio que contiene 1189 errores encontrados en un experimento realizado en una universidad referente al desarrollo de programas de pequeña extensión, escritos en FORTRAN, PL/1, ALGOL, BASIC y COBOL**. Por otro lado, la IBM tiene una base de datos que contiene las fallas que han tenido sus equipos en operación y la manera en que se han solucionado.

Bajo el rubro de herramientas automáticas dinámicas para el proceso de probar el software se tiene:

Generador de archivo de pruebas. Un generador de archivo de pruebas (test file generator TFG) crea un archivo de información con ciertas características deseadas por el usuario, que sirve como entrada al programa que va a ser probado. Como se mencionó con anterioridad, los casos de prueba deben estar compuestos por las entradas y las salidas esperadas; en este caso el generador sólo produce las entradas y de manera casi aleatoria, por lo que no se puede considerar una herramienta de calidad. Aunque cuando está realizán-

* An Analysis of Errors and their Causes in System Programs, A. Endres, IEEE Transactions Software Engo, SE-1(2), 140-149 (1975)

** Human Errors in Programming, E.A. Youngs, Int. J. Man-Machine Stud. 6(3), 361-376 (1974) dose una prueba donde no interesen mucho las salidas (p.e. pruebas de volúmen) es de gran utilidad.

El equipo UNIVAC 1100/82 tiene un programa de utilería llamado TFG que produce este tipo de archivos, teniendo entre otras posibilidades determinar su organización, su longitud, su contenido y su tamaño.

Generadores de datos de Prueba. En la actualidad aún no se han desarrollado generadores de casos de prueba con sus respectivas entradas y salidas, lo más que se ha podido lograr es realizar programas que analicen el flujo de control de un programa y propone un conjunto de entradas de acuerdo a un determinado criterio de la técnica de caja blanca.

Generador de impulsores de código. Como se mencionó anteriormente, cuando está realizándose la prueba de integración o de módulos, es necesario contar con los módulos impulsores (aquél que invoca el módulo a probar), los cuales son escritos para alimentar los casos de prueba al módulo. Las herramientas para la generación de impulsores de código, proporcionan un substituto y proporcionan un lenguaje sencillo para expresar los casos de prueba.
Con esto se logran ventajas como: a) facilidad en la ejecución de los casos de prueba, b) estandarización en los casos de prueba, y c) verificación automática cuando se ejecuta el programa, ya que se proporcionan tanto las entradas al programa como las salidas esperadas (casos de prueba).

En la actualidad existe una herramienta bastante sofisticada de este tipo llamada unidad de prueba automatizada (Automated Unit Test AUT). Para usar AUT se codifican casos de prueba en un lenguaje que no es el de un procedimiento y los almacena en una base de da-

El módulo se prueba operando un único comando desde una terminal. Esto hace que AUT compile cada caso de prueba, ejecute el módulo con los datos de entrada apropiados, compare los resultados obtenidos con los esperados y notifique cualquier diferencia al operador*.

Comparador de salida. El comparador de salida es un programa sencillo que compara un par de archivos localizando sus diferencias. Un archivo contiene los resultados esperados, esto es, las salidas esperadas de los casos de prueba. El otro archivo contiene los resultados generados por el programa probado. Esta herramienta es utilizada para pruebas de cualquier tipo, es de gran utilidad ya que provee un método de comparación automático que elimina la tediosa comparación manual, obteniéndose de esta manera reducción de tiempos y costos.

La combinación adecuada de las herramientas automáticas para el proceso de probar descritas previamente, pueden incrementar los beneficios, ya que podría lograrse que el proceso fuera prácticamente automático, ya que se contemplan herramientas que pueden ser utilizadas con la siguiente secuencia:

+ Analizador estático de código que proporcione un análisis del código fuente.

+ Recopilador de errores para asistir en el diseño de los casos de prueba.

* Automated Test and Verification, C.A. Heuermann, G.J. Myers, and J.H. Winterton, IBM Tech. Disclosure Ball. 17(7), 2030-2035 (1974)

Automated Unit Test (AUT) Program Description/Operation Manual SH20-1663. White Plains, N.Y. IBM, 1975

Automatic Software Test Drivers, D.J. Panzl, Computer 11(4), 44-50 (1978)

+ Biblioteca de código reusable para la utilización de los casos de prueba.

+ Generadores de datos y archivos de prueba para la creación de datos de entrada para el programa a probar.

+ Impulsores de código para la ejecución de las pruebas del tipo modular y de integración, así como la comparación de resultados.

+ Comparador de salida para generar un reporte de las diferencias entre las salidas propuestas y las obtenidas.

Además, la ejecución de toda esta combinación puede ser automatizada con lo que se crearía una nueva herramienta automática que se encargará de dicha ejecución.

Puede esperarse que para abatir costos y tiempos en el proceso de probar se tienen que diseñar herramientas automáticas más sofisticadas que tengan una mejor interface hombre-máquina, que sean inteligentes y que tengan una menor dependencia hacia un sistema o metodología en particular.

Otro aspecto muy importante sería la creación de un generador de casos de prueba (con sus respectivas entradas y salidas), a partir de los requerimientos y del programa, tarea bastante difícil.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

ix. LIBERACION DEL SISTEMA
(DOCUMENTACION BASICA)

FEBRERO, 1984

PANORAMA GENERAL

FASES FINALES DE PROYECTOS INFORMATICOS

- LIBERACION DE SISTEMAS INFORMATICOS 1
- SISTEMAS EN PRODUCCION 2
- MANTENIMIENTO DE SISTEMAS 3

1 LIBERACION DE SISTEMAS

- VENTA DEL PRODUCTO Y PREPARACION
  - Entusiasmo
  - Beneficios
  - Estrategia de liberación
    - DEFINICION
    - Lista de una estrategia de liberación
  - PROCEDIMIENTOS Y CONTROLES DEL USUARIO
    - DEFINICION
    - Consecuencias de ignorar los procedimientos
    - Responsabilidad de la definición de procedimientos
    - Proceso de definición de procedimientos
    - Lista de procedimientos comunes y características
    - Problema de los manuales de procedimiento
- ENTRENAMIENTO AL USUARIO
  - DEFINICION
  - Proceso de entrenamiento
  - Técnicas utilizadas para el entrenamiento
  - Métodos de entrenamiento
  - Retroalimentación del usuario
  - Reciclaje de entrenamiento del usuario al usuario
  - Recomendaciones importantes para el entrenamiento
- CONVERSION DE SISTEMAS
  - DEFINICION
  - Actividades principales
  - Factores delicados en la conversión
  - Consideraciones importantes para el éxito de la conversión
- PRUEBAS DE ACEPTACION
  - ENFOQUE DE LAS PRUEBAS DE ACEPTACION
    - DEFINICION
    - Consideraciones importantes del enfoque de pruebas de aceptació
  - CRITERIOS DE ACEPTACION DE PRUEBAS
    - Operativos
    - Técnicos
  - CARACTERISTICAS DE LAS PRUEBAS DE ACEPTACION
  - TECNICAS DE LAS PRUEBAS DE ACEPTACION
  - CONSIDERACIONES IMPORTANTES EN EL CASO DE PRUEBAS

2 **SISTEMAS EN PRODUCCION**

- DEFINICION REVISIONES
  - CON RESPECTO AL USUARIO
  - DESEMPEÑO DE HARDWARE/SOFTWARE
- MEJORAS/CAMBIOS A SISTEMAS EN PRODUCCION
  - DEFINICION
  - IMPACTO
  - RECOMENDACIONES

3

MANTENIMIENTO DE SISTEMAS

DEFINICION

FACTORES DELICADOS

CAUSAS POR LAS QUE EL MANTENIMIENTO ES NECESARIO

ADMINISTRACION DEL MANTENIMIENTO (RECOMENDACIONES)

DESARROLLO DEL MANTENIMIENTO

VENTA DEL PRODUCTO Y PREPARACION DE LA LIBERACION

- ENTUSIASMO
- BENEFICIOS
- ESTRATEGIA DE LIBERACION
  - DEFINICION
  - LISTA DE UNA ESTRATEGIA DE LIBERACION
- PROCEDIMIENTOS Y CONTROLES DEL USUARIO
  - LIBERACION
  - CONSECUENCIAS DE IGNORAR LOS PROCEDIMIENTOS
  - RESPONSABILIDAD DE LA DEFINICION DE PROCEDIMIENTO
  - LISTA DE PROCEDIMIENTOS Y CARACTERISTICAS
  - PROBLEMAS DE LOS MANUALES DE PROCEDIMIENTO

LIBERACION DE SISTEMAS INFORMATICOS

VENTA DEL PRODUCTO

La culminación del esfuerzo de desarrollo se lleva a cabo en la fase de liberación del sistema; consideramos que el tratamiento adecuado de esto requiere; ¡mucho entusiasmo!, por parte del equipo de desarrollo, porque de nada valdría todo el esfuerzo hecho, si no de realiza, una buena VENTA DEL PRODUCTO, y esto lo decimos en relación a destacar los beneficios a la organización, al personal usuario, al personal operativo y al equipo de desarrollo del sistema, quien no debe decaer en su esfuerzo por el éxito.

Todo lo anterior, para que se realice, debe estar cuidadosamente planeado y sin lugar a dudas, el énfasis en las relaciones internas y externas al sistema deben conservarse y cuidarse más que en ninguna otra parte del ciclo de desarrollo.

PREPARACION

Para lograr lo anterior procederemos en principio a preguntarnos.

Estoy preparado para la liberación ?

Definitivamente la pregunta anterior nos llevará al establecimiento de una ESTRATEGIA DE LIBERACION.

Consistiendo esta, en la definición clara y objetiva del conjunto de actividades y apoyos que requiere la liberación del sistema, un esquema de los principales temas que debe cubrir esto lo mostraremos en el diagrama siguiente.

Vemos un énfasis fundamental en las pruebas de aceptación y todo lo que esto involucra, tales como instalación de Hardware/Software especializados, consideraciones como redistribución del personal para el desarrollo de nuevas actividades, representación del lugar en donde se llevará a cabo la liberación, entrenamiento del usuario, y en si, todos los elementos necesarios para dar por concluído el desarrollo del sistema.

SE DEBE HACER:

LISTA DE UNA
ESTRATEGIA
DE
LIBERACION

(Definición clara y objetiva del conjunto de actividades y apoyos para la liberación)

- Revisión de procedimientos y controles
- Entrenamiento al usuario
- Calendario de liberación de los múltiples segmentos del sistema
- Calendario de liberación de localizaciones múltiples
- Necesidad de un lugar para una prueba piloto
- Pruebas en paralelo entre el viejo y el nuevo sistema
- Criterios de aceptación del nuevo sistema
- Planes de instalación de Hardware/Software
- Enfoque para eliminar por fases las operaciones actuales y redistribuir el personal
- Definir los compromisos y costos de la liberación
- Consideraciones para la carga de la Base de Datos y operaciones en linea (conversión)
- Personal responsable para establecer contactos y los calendarios que deben coordinarse

Por otra parte, la estrategia de liberación sirve para llevar a cabo el cónjunto de actividades de indole administrativo (planear, organizar, dirigir, controlar y evaluar), así como, reunir el conjunto de documentos necesarios para integrar dentro del expediente del sistema, los eventos y actividades que facilitan la ejecución de esta fase.

Los objetivos generales de la estrategia de liberación se ven aplicados en los diferentes planes que se elaboran en cada una de las etapas que integran esta fase, existiendo desde el inicio, objetivos como los descritos en la lista anterior, siendo obvio que si alguno de estos, no se considerase; entonces, provocaría conflictos con graves riesgos. A través de las siguientes páginas se encontrarán una serie de actividades que muestran mas objetivamente como aplicar las estrategias; en las actividades críticas de esta fase, encontraremos respuestas concretas a las preguntas, qué se debe hacer ?, como se hace ?, y qué debemos cuidar?.

Por supuesto, ahí que considerar que todo lo anterior tiene un ambiente relacionado, es decir, si el proyecto se consideró en un principio centralizado, su liberación debe ser igual, si fué desconcentrado deberá dirigirse a este objetivo. En el caso de que se tratara de grandes equipos; tal vez, habría que adquirir oportunamente el Hardware necesario para la nueva aplicación, y lo mismo ocurriría si se tratara de un ambiente basado en minicomputadoras o computadoras personales.

Cabe finalmente la aclaración, que la estrategia mostrada es útil, según: "el cristal con que se mira", como un reforzamiento a lo anterior expuesto. Las estrategias son una guía para la acción.

Sin embargo, pese a haber definido nuestra estrategia de liberación, consideramos que existen todavía respuestas a la pregunta.

ESTOY PREPARADO PARA LA LIBERACION ?

Y es evidente que pese a conocer que quiero hacer para la liberación como punto de partida importante; debemos, como equipo de desarrollo de sistemas informáticos, preocuparnos por tener preparado todo el instrumental necesario para que esta fase no falle; por esto, después de definir - las estrategias de liberación, el paso siguiente es tener claridad en los PROCEDIMIENTOS Y CONTROLES DEL USUARIO, estos tienen como objetivo, asegurar que la operación del sistema sea adecuada, lo que implica que el usuario debe conocer con precisión cuales serán sus responsabilidades, lo que debe hacer durante la ejecución del sistema, y además mostrar ayudas suficientes y necesarias ante cualesquier eventualidad que se presente tales como errores, revisiones y toda clase de soportes que faciliten el uso y operación del sistema; hemos observado que en muchas ocasiones , la causa para que un usuario no acepte el sistema se debe a que el equipo de desarrollo, no hizo énfasis en el conjunto de PROCEDIMIENTOS Y CONTROLES de uso del sistema, resumiremos en el diagrama siguiente las consecuencias que esto implica.

Ingorar los procedimientos y controles tiene como consecuencia

- Falta de definición de responsabilidades
- Entrenamiento inadecuado
- Deficiencia de respuestas ante problemas que pudieran presentarse

RESPONSABILIDAD DE LA DEFINICION DE PROCEDIMIENTOS Y CONTROLES

La responsabilidad de la definición de los procedimientos y controles del usuario, tiene las siguientes jerarquías.

1) Equipo de desarrollo

Dado que estos son quienes conocen perfectamente el desarrollo del proyecto, desde los requerimientos iniciales y todo lo que en consecuencia se produjo para satisfacer la necesidad, sin duda alguna, si el personal de desarrollo no se involucra en la definición, las implicaciones son graves.

2) Auxilio del usuario

Por supuesto que nadie mejor que el usuario conoce realmente su problemática, y en este sentido, quien mejor puede sugerir, que tipo de procedimientos necesita es el mismo; también, puede señalar cuales se le hacen complicados o dificiles de aplicar, con esto el equipo baja a la tierra y ve lo que realmente se necesita.

3) Particinación de auditores

Es recomendable que para evitar la ausencia de normas de auditoría, los auditores participen con las revisiones y sugerencias que mejoren la calidad del trabajo.

4) Auxilio de unidades de métodos y nrocedimientos

En ocasiones si se cuenta con el apoyo organizacional de unidades de métodos y procedimientos, la ayuda que pudieran prestar al desarrollo de procedimientos y controles sería invaluable, dado que en estas áreas se - concentra una experiencia acumulada bastante amplia, - que daría como resultado una precisión y respeto claros a las normas que la organización maneja.

Como consecuencia de conocer quien hace que, el paso siguiente razonablemente debe ser como me preparo para hacer los procedimientos, esto da origen al proceso de definición de procedimientos y controles.

PROCESO DE DEFINICION DE PROCEDIMIENTOS Y CONTROLES

El proceso para lograr una definición correcta de procedimientos, tiene fundamentalmente cuatro actividades principales.

1) Preparación de un plan de trabajo

Basicamente lo que busca definir tiempos; recursos y responsabilidades para obtener eficazmente los procedimientos, es importante que en esta actividad se considere la definición de los estandares a usar, así como el conjunto de revisiones a efectuar en el transcurso de esta actividad.

2) Revisión del desarrollo del proyecto

Esta actividad tiene fundamentalmente dos aspectos a cubrir:

2.1 Revisión de los requerimientos del sistema. Donde se observan las funciones, flujo de información de las áreas usuarias, la seguridad y principalmente los cambios en la organización, dado que no se puede hacer referencia a algo que no existe.

2.2 Revisión de especificaciones técnicas

En esta actividad lo que debemos estudiar y tener claro es la forma en que fueron dispuestos los siguientes puntos.

- Diálogos y Menús
- Reglas de corrección
- Mensajes de error
- Claves de acceso
- Cifras de control
- Bitácoras para auditoría
- Protección y recuperación

Cuando terminamos de revisar el desarrollo del proyecto, se debe tener comprensión total de los conceptos y especificaciones.

3) Definición de estandares de elaboración

Nosotros vemos importante el definir un método y una estructura básica para efectuar desarrollos futuros, si esto se tiene, es recomendable que los estandares de procedimientos describan claramente las funciones y subfunciones, tareas o actividades mediante encabezados precisos de lo que se quiere hacer, ademaś no se debe olvidar la definición del respon sable usuario, los pasos que se tienen que efectuar y la definición clara del próximo responsable.

4) Estructura de un manual

También, si no se tiene, es recomendable integrar todos los procedimientos definidos en un manual que facilite el acceso a la información que el sistema proporciona destacando minimamente los siguientes puntos:

- Indice
- Políticas
- Normas
- Base Legal
- Panorama general
- Descripción de procedimientos
- Responsabilidad de corrección y actualización del sistema

Y finalmente algo que generalmente se olvida

- Responsable del mantenimiento del manual, para evitar olvidos involuntarios.

Como un resumen de lo anterior mostraremos el siguiente diagrama, y a continuación realizaremos un análisis de los principales procedimientos.

PROCESO DE DEFINICION DE PROCEDIMIENTOS Y CONTROLES

- Preparación de un plan dc trabajo
- Revisión del desarrollo del proyecto
  - Revisión de los requerimientos
    - <u>Se debe observar:</u>
      - Funciones
      - Flujo de información de áreas usuarias
      - Seguridad
      - Cambios en la organización
  - Revisión de especificacionés técnicas
    - <u>Se debe observar:</u>
      - Diálogos y menús
      - Reglas de corrección
      - Mensajes de error
      - Claves de acceso
      - Cifras de control
      - Bitácoras de auditoría
      - Protección y recuperación
- Se tiene una comprensión total de los conceptos y especificaciones
- Definición de estandares de elaboración (0,1)
  - Utilizar encabezados claros para definir funciones, subfunciones ect.
  - Definir responsable
  - Descripción de pasos
  - Definición clara de la la siguiente función o responsable
- Definición de la estructura de un manual (0,1)
  - Indice
  - Políticas
  - Normas
  - Base legal
  - Panorama general
  - Descripción de procedimientos
  - Responsabilidad de corrección y actualización del sistema
  - Responsabilidad el mantenimiento del manual para evitar olvidos.

PROCEDIMIENTOS MAS COMUNES

Como parte integrante de la preparación para liberar el sistema, debemos recalcar, que sin los procedimientos adecuados no se tendría ningún éxito. Esto implica tener claridad en los - procedimientos que se definen y dentro de los mismos los puntos que merecen mayor atención para efectuarlos como corresponde.

Utilizaremos una tabla descriptiva dividida en dos columnas - que faciliten la comprensión de los procedimientos; la primer columna describe conceptualmente el procedimiento y la segunda destaca los aspectos mas importantes que debe cubrir.

Procedimiento	Aspectos importantes
Entradas/salidas	- Clase de codificación
	- Agrupación en lotes
	- Cáptura
	- Periodicidad
	- Cifras de control
	- Totales de reportes
	- Control de correcciones
	- Reenvío de rechazos
	- Solicitudes especiales.

Para controlar eficientemente este procedimiento se puede hacer uso de : fecha, sellos fechadores y documentos prenumera-

Prodedimientos	Aspectos importantes
Archivo de Datos/ Base de Datos	- Cuidar la integridad de tal forma que los archivos correspondan a las corridas requeridas. - Recuperaciones y respaldos periodicos.

Para efectuar un control de dicho procedimiento es recomendable contar con; cifras control de registros existentes, totales combinados de variables, principalmente delicadas.

Procedimiento	Aspectos importantes
Controles de acceso	- El propósito de esto consiste en restringir el acceso a programas del computador, sin la debida autorización. - Se debe de; controlar la expedición de claves de acceso. - Aplicarlo a reportes confidenciales de cheques y datos confidenciales de personal.

Procedimiento	Aspectos importantes
Actualización de archivos	- Calendario de aplicación de transacciones. - Manejo de versiones programadas para filtrado que varíen con el tiempo. - Separación de responsabilidades.

Procedimiento	Aspectos importantes
Instrucciones técnicas de operación	- Describen el conjunto de ayudas para que el operador utilice eficientemente el sistema.
	- Incluye ayudas como: Muestras de display y decisiones tomadas por el operador para responder adecuadamente.
	- Se deben considerar la limpieza y mantenimiento del lugar de trabajo.
	- En el caso de manejo de papelería estandard y especializada es necesario precisar claramente cuando se debe usar.
	- Para el manejo de cintas y manejo de discos removibles, es conveniente respetar los procedimientos de etiquetado automático y manual, señalando las responsabilidades de manejo.

Finalmente es conveniente recordar cuales son los <u>problemas típicos de los manuales de procedimientos</u>.

- Incapacidad para localizar con rapidez las respuestas a un problema
- Incapacidad para entender las respuestas proporcionadas.
- Incapacidad para determinar acciones, una vez que se han identificado los problemas
- Exceso de detalle en una tarea sencilla
- Formatos poco práticos, que requieren muchas páginas para describir una sola función
- Falta de relación con el todo del sistema.

Para abatir este tipo de problemas debemos auxiliarnos de los mejores instrumentos existentes, tales como los diagramas de funciones y flujo de datos y mejorarlos con el material expuesto, lo importante para hacer bien esta actividad es no olvidar que se trata de un sistema y este interactúa con varias partes, pues de esta manera se logra una comprensión total y no aislada.

ENTRENAMIENTO AL USUARIO

Para efectuar una correcta liberación, debemos además de proporcionar los procedimientos y controles precisos, comunicar al usuario el uso correcto del sistema; esta actividad, cuando la realizamos debemos hacerla con el propósito de preparar al usuario para la prueba de aceptación y recibir retroalimentación que mejore el sistema.

Para esto es importante comenzar seleccionando el lugar, y al personal necesario; basados en un programa de entrenamiento, consideramos aquí, que el material de apoyo se encuentra principalmente en los procedimientos; en esta fase tenemos que tener una voluntad muy grande para que la venta del sistema se realice, poniendo todo el esfuerzo posible de nuestra parte, tal vez más que en ninguna otra fase.

Además de lo anterior es claro que los entrenadores deben estar bien preparados; en este sentido, es conveniente que se vaya logrando una especialidad de esta actividad a tal grado que inclusive se genere un equipo, que entienda de pedagogía y didáctica, ademas, en buena parte la -- conciencia del vendedor.

Sintetizando el proceso de entrenamiento lo mostramos en el siguiente diagrama.

PROCESO DE ENTRENAMIENTO

- Definir un programa de entrenamiento organizado
- Seleccionar el sitio y personal de entrenamiento
- Seleccionar el grupo de entrenamiento, bien preparado
- Preparar el material de apoyo
- Contando con los manuales de procedimientos, esto ya se realizó.

Para una efectiva comunicación los entrenadores deben hacer uso de todas las técnicas posibles incluyendo, desde un muy sencillo rotafolio, hasta enseñanza (si es posible) por computador.

A continuación resumimos las principales técnicas usadas en el entrenamiento, llendo desde las mas simples a las mas complejas.

TECNICAS UTILIZADAS PARA EL ENTRENAMIENTO

- Pizarrón y gis (con buena voluntad)
- Rotafolio
- Diapositivas
- Transparencias
- Videograbación
- Utilización del computador para la enseñanza

El entrenador debe, por otra parte auxiliarse de los métodos de entrenamiento, para lo cual haremos una síntesis en algunos métodos.

METODOS DE ENTRENAMIENTO

- Demostraciones
- Estudio de casos prácticos
- Explicación de conceptos básicos
- Trabajo en equipo
- Reenseñanza del usuario

El propósito de entrenar es aprender a usar, y también pese a todo nuestro esfuerzo, obtener la RETROALIMENTACION DEL USUARIO, ya que es esta la principal razón de haber hecho el sistema, porque si un sistema no tiene características humanas en lugar de ser una ayuda pasa a ser una carga.

La retroalimentación del usuario, pretende incluir mejoras/ cambios que den mejores respuestas a los usuarios, sin embargo no es posible descontar que hubiese fallas; también es necesario tener un control de calidad del entrenamiento; es decir, debemos medir el interés en las sesiones, la presentación de los instrumentos utilizados, el entendimiento del material, y la solución correcta para respuestas a problemas específicos, todo esto nos ayudará a preparar mejor el sistema para la etapa de pruebas de aceptación.

Por otra parte, es importante tratar de que la actividad de entrenamiento sea responsabilidad del propio usuario, a esto lo denominamos RECICLAJE DE ENTRENAMIENTO DEL USUARIO AL USUARIO, cuando no se tiene una concepción clara o simplemente no se mencionó esto, el efecto nunca se realiza; sin embargo, si los desarrolladores de sistemas desean dedicarse más a la tarea que tienen que efectuar es necesario introducir este concepto, por supuesto esto no se realiza facilmente en organizaciones que comienzan con sistemas automatizados; por lo que muchas veces este concepto no siempre tiene éxito, lo que es importante es ir introduciendo esto y simplemente con el transcurso del tiempo esta disciplina se puede adquirir. Es necesario por lo tanto dos conceptos importantes.

RECICLAJE DE ENTRENAMIENTO DEL USUARIO AL USUARIO

1) Definir futuros instructores del mismo usuario

2) Responsabilizar al usuario del mantenimiento del material de entrenamiento.

Para que el entrenamiento tenga un desarrollo efectivo y se logren todos los propósitos esperados, es importante destacar las siguientes recomendaciones, mostradas a continuación.

RECOMENDACIONES
IMPORTANTES
PARA EL
ENTRENAMIENTO

- Enseñar todo el sistema

  De esta manera se evitan malos entendidos y se asegura que la responsabilidad se asuma completamente en las áreas del usuario.

- Evitar detalles técnicos y palabras sofisticadas de computación

  Esto logra una comunicación exitosa sin bloqueos mentales

- Claridad de objetivos

  Sin esto el usuario no comprende su rol en el sistema

- Venta del sistema

  Demostrar los beneficios que proporciona el sistema, insistiendo tantas veces como sea necesario

- Relajar la ansiedad

  Atenuar en lo posible la resistencia al cambio.

CONVERSION DE SISTEMAS

- DEFINICION
- ACTIVIDADES PRINCIPALES
- FACTORES DELICADOS EN LA CONVERSION
- CONSIDERACIONES IMPORTANTES PARA EL EXITO DE LA CONVERSION

CONVERSION DE SISTEMAS

El objetivo de esta etapa dentro del contexto de liberación del sistema y en correspondencia a las estrategias de liberación; consiste en: aprovechar la información existente, considerar los cambios a datos que no estaban disponibles o tenían inconsistencias; esta etapa por estrategia debe realizarse antes de las pruebas de aceptación, debido a que en las pruebas requerimos de información convertida.

Las actividades principales que se llevan a cabo son:

1) Definición de programas de conversión

2) Construcción de programas de conversión

3) En paralelo a los puntos anteriores, debemos definir la manera de adquirir archivos correctos. Los archivos correctos se pueden adquirir mediante dos clases de esfuerzo.

3.1) Esfuerzo de purificación de datos.

Lo cual consiste en corregir, ampliar o adaptar datos existentes.

3.2) Esfuerzos dc convcrsión manual.

Es decir, la adquisición de datos no existentes, que ahora en el nuevo proceso son necesarios.

Es posible en esta etapa utilizar programas producto de uso generalizado, que faciliten la conversión, tales como: formateadores, clasificadores y super editores; así como, programas orientados a cambios de código. Este tipo de herramientas hacen posible que el trabajo de conversión se acelere en un tiempo mínimo.

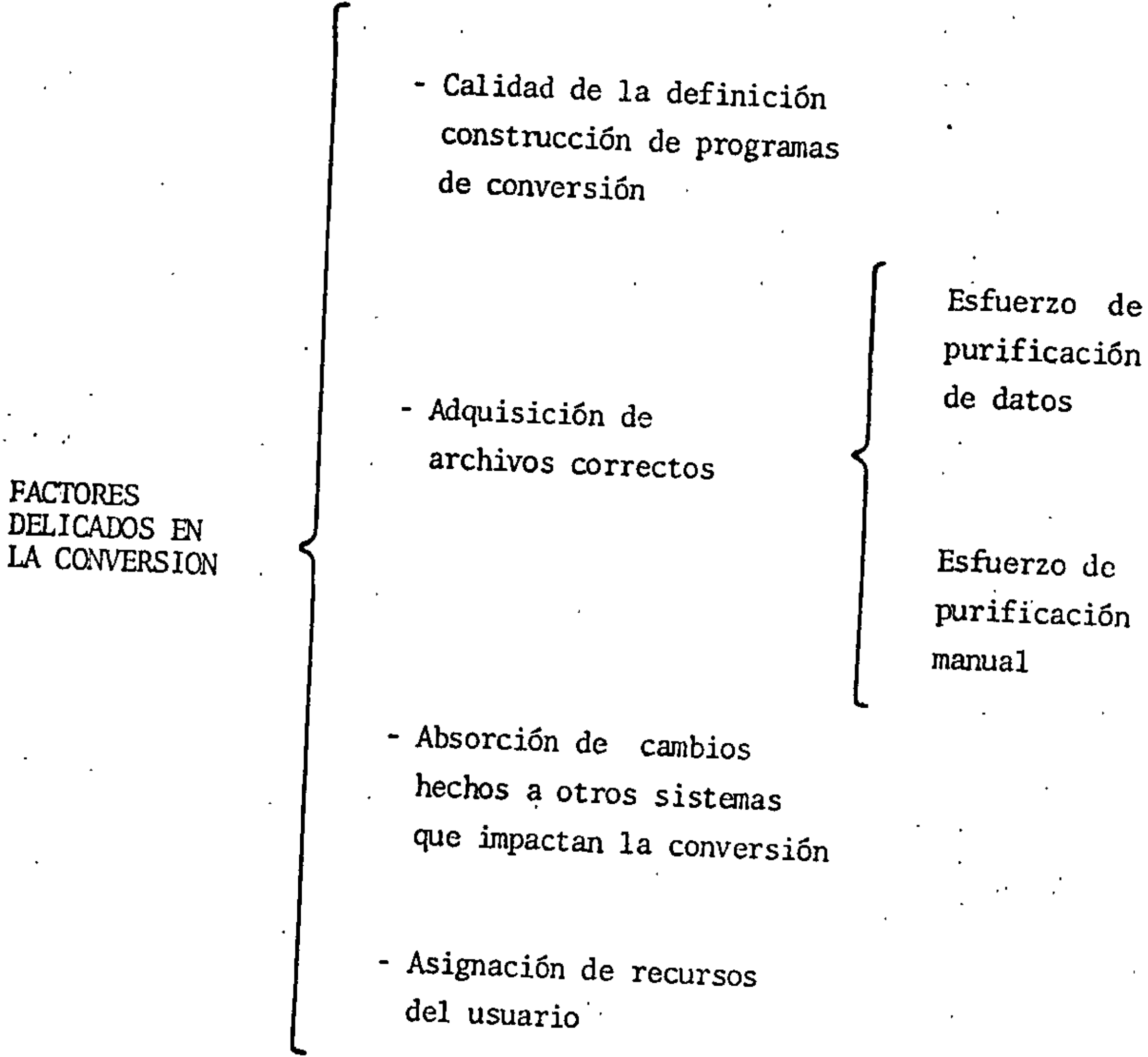
Sin embargo, si esto no fuera posible de utilizarse, es recomendable que todo tipo de programas se realice con las técnicas estructuradas, con una revisión mucho mas minuciosa ya que en un error realizado en la conversión, impacta a tal grado que evita pasar a la siguiente etapa y en ocasiones un error de conversión, causa que el sistema se suspenda.

Además de cuidar la calidad de la producción de los programas de conversion es recomendable adicionar un alto control de calidad a los datos convertidos o adicionados, esto puede lograrse mediante la inclusión de programas que chequen la integridad de datos, esto se hace igualando los datos anteriores contra los convertidos, ya sea mediante pequeñas muestras o todo lo convertido; indicando que de encontrarse alguna diferencia, esto obliga a suspender el proceso, implicando una revisión y corrección de errores.

Con el transcurso del tiempo, los sistemas son tan dinámicos, que pese a estar un sistema en desarrollo, se siguen haciendo cambios, en la vieja versión del sistema, esto implica que la conversión realizada puede quedar obsoleta, por lo que se recomienda, hacer cortes periodicos a la información existente y en un momento dado volver a convertir la información tantas veces como sea necesario, hasta empatar la última versión.

Otro punto de interés que no puede pasarse por alto, es la asignación de recursos del usuario a esta etapa, ya que si este hecho se ignora, no va a existir ninguna responsabilidad para agregar nuevos datos al nuevo sistema, o corregir los ya existentes, teniendo como consecuencia un doble esfuerzo para el área informática, es necesario que de antemano el usuario tenga el pleno y absoluto conocimiento de esta responsabilidad.

Como un resumen de los factores mas delicados de la conversión utilizaremos el siguiente diagrama



FACTORES DELICADOS EN LA CONVERSION

- Calidad de la definición construcción de programas de conversión
- Adquisición de archivos correctos
  - Esfuerzo de purificación de datos
  - Esfuerzo de purificación manual
- Absorción de cambios hechos a otros sistemas que impactan la conversión
- Asignación de recursos del usuario

Del mismo modo que existen un conjunto de factores que no debemos descuidar, existen también un conjunto de consideraciones que son importantes para el éxito de la conversión, lo cual resumimos a continuación.

Consideraciones importantes para el éxito de la conversión.	- Problema de mantenimiento paralelo de archivos	Se debe buscar que los archivos del antiguo y el nuevo sistema no se encuentren juntos, esto puede lograrse usando tiempo bloque
	- Coordinación con la prueba y el entrenamiento	La etapa de conversión no debe tener conflictos con los planes de prueba y entrenamiento, es necesario hacer esto de acuerdo a las posibilidades del usuario
	- Calidad de los datos	Se debe definir níveles mínimos de aceptación tanto de los esfuerzos manuales como los esfuerzos automáticos
	- Conversión de datos históricos	Por olvido o por descuido en ocasiones no se considera la conversión de la información histórica, sin embargo, para estudios futuros, esta es esencial
	- Adecuar la conversión a la capacidad de la organización	No es posible exigir al usuario, personal que no se tenga contratado. O bien, si los recursos son pocos contratar horas extras.

PRUEBAS DE ACEPTACION

- ENFOQUE DE LAS PRUEBAS DE ACEPTACION
  - DEFINICION
  - CONSIDERACIONES IMPORTANTES DEL ENFOQUE DE PRUEBAS DE ACEPTACION
- CRITERIOS DE ACEPTACION DE PRUEBAS
  - OPERATIVOS
  - TECNICOS
- CARACTERISTICAS DE LAS PRUEBAS DE ACEPTACION
- TECNICAS DE LAS PUEBAS DE ACEPTACION
- CONSIDERACIONES IMPORTANTES EN EL CASO DE PRUEBAS

ENFOQUE DE LAS PRUEBAS DE ACEPTACION

Es una prueba a nivel sistema, con el propósito de encontrar errores y corregirlos, esto incluye; programas de computadora, procedimientos y controles, manuales y automáticos, conversión de datos, entrenamiento al usuario y al personal operativo de la unidad de informática.

El enfoque principal es el de sistemas, es decir, debe observarse la solución en forma total y no parcialmente, esto va, desde el mas sofisticado porgrama automatizado que resuelve de "todo", hasta las previsiones hechas para que -- existan, papel de impresión, impresoras, terminales, cintas y discos.

La prueba de liberación o de aceptación, pese a la urgencia que la institución tenga para hacer uso del sistema tienen que realizarse en estricto respeto a un plan, fundamentalmente se debe probar que el sistema cumpla con los objetivos - para los que fué desarrollado y no falle cuando se suciten eventos no esperados.

Deben asignarse adecuadamente los recursos, técnicos, materiales y de personal, adecuados para llevar a cabo las corridas de prueba, debe existir disponibilidad de dispositivos, los materiales de impresión y respaldo se deben tener oportunamente, así como el personal del usuario y técnico para que chequen los resultados, y corrijan programas, procedimientos, controles y formas que se afecten.

Como en todas las fases anteriores, para hacer bien las cosas debemos estar preparados.

Un punto delicado que debemos considerar es la cantidad de tiempo adecuado para permitir que las pruebas se hagan en una secuencia continua, hasta que las condiciones hayan sido checadas y corregidas, y todos los participantes estén satisfechos con las nuevas operaciones.

Las pruebas deben ser cordiales, es decir, de ninguna manera se debe relajar el respeto al usuario; principalmente -- cuando expresa opiniones y sugerencias del producto, una actitud agresiva provoca una falta de interés, y si deseamos que el producto sea aceptado nada mejor que entender la retroalimentación.

CONSIDERACIONES IMPORTANTES DEL ENFOQUE DE PRUEBAS DE ACEPTACION.

- LA PRUEBA ES A NIVEL SISTEMA Y NO POR PARTES, SE DEBEN ENCONTRAR ERRORES DE VOLUMEN Y RESISTENCIA, Y CORREGIRLOS.
- EN ESTA PRUEBA SE TRATA DE REVISAR, DESDE EL MAS SOFISTICADO PROGRAMA, HASTA LOS RECURSOS HUMANOS, Y MATERIALES BASICOS.
- LAS URGENCIAS DEBEN RESPETAR UN PLAN, Y ESTE DEBE CORRESPONDER A LOS OBJETIVOS, ASI COMO PREVER SOLUCIONES A CONFLICTOS.
- LOS RECURSOS TECNICOS, MATERIALES, HUMANOS, DEBEN EXISTIR OPORTUNAMENTE, CON LAS RESPONSABILIDADES CORRECTAMENTE DEFINIDAS.
- EL TIEMPO DEBE CORRESPONDER EXACTAMENTE A UNA SECUENCIA CONTINUA, HASTA QUE TODAS LAS CONDICIONES SE HAYAN CUMPLIDO, Y POR SUPUESTO QUE LAS NECESIDADES SE CUMPLAN.
- SE DEBE MANTENER CORDIALIDAD EN LAS RELACIONES CON EL USUARIO, DE NINGUNA MANERA NEGAR LA RETROALIMENTACION.

CRITERIOS DE ACEPTACION DE SISTEMAS

Después de establecido un enfoque para el proceso de pruebas de aceptación, se debe definir conjuntamente con el - usuario, ¿cuáles son los criterios de aceptación del sistema?, esto implica que se consideren todos los puntos - necesarios para dar por concluido el proceso de pruebas - de aceptación.

En esta etapa debe hacerse un refrescamiento de todos los requerimientos, tanto operativos como técnicos, que el - usuario haya solicitado o debió haber solicitado para la aceptación del producto.

Por operativos, consideraremos el conjunto de auxiliares que facilitan el desarrollo del trabajo, tales como ayudas automáticas, procedimientos, presentación de reportes o pantallas, o bien que cumpla con un ciclo preestablecido, en tiempo y en lugar.

Con respecto a los criterios técnicos, debemos establecer, criterios de eficiencia, tales como: tiempos de respuesta aprobados en la etapa de requerimientos, métodos de recuperación y respaldo, claramente establecidos, tipos de datos que deben existir y, manejarse en forma adecuada por el sistema, criterios de validación, actualización, y explotación, correctamente realizados.

CRITERIOS DE ACEPTACION DE SISTEMAS

- OPERATIVOS
  - Todos los resportes especificados en los requerimientos del usuario se producen y se pueden usar.
  - Las formas y procedimientos de control manual pueden utilizarse y cumplen con las funciones que se espera de ellos.
  - Los manuales de procedimientos son faciles de utilizar y describen con precisión, reportes, procedimientos, uso de pantallas y corrección de errores.
  - El sistema fluye a través de un ciclo dentro de las limitaciones especificadas.
- TECNICOS
  - Tiempos de respuesta para varios tipos de transacciones, dentro de los límites de tiempo prescritos, de acuerdo a los requerimientos.
  - Tipos de datos que deben existir y manejarse en forma adecuada por el sistema.
  - Todos los métodos y procedimientos de recuperación y respaldo, operan correctamente.

CARACTERISTICAS DE LAS PRUEBAS DE ACEPTACION

La realización de las pruebas implica, probar eficaz, eficientemente, y efectivamente, todos los componentes del sistema. Esto incluye todos los procedimientos computarizados y manuales de manera integrada.

Para efectuar las pruebas, se requiere; haber realizado una prueba previa de los módulos por separado, y después, integrados. Es recomendable que los cambios detectados en el entrenamiento y en la conversión se hayan incorporado. La idea principal de esta clase de pruebas, es probar; en volumen y resistencia el sistema, solo de esta manera es posible demostrar que se desempeñará correctamente.

Una analogía importante de este concepto es probar el motor completamente, y no pensar, que si las partes del motor funcionan, entonces el motor funciona.

Se puede considerar esta fase como una simulación del trabajo real que el sistema desempeñará, con las condiciones diarias que se le exijan y las tensiones que se sucitan durante las operaciones regulares, incluyendo; tensiones técnicas, como apagones simulados y destrucción de archivos.

Como punto de inicio, se debn preparar, tanto datos de prueba, como una lista de chequeo de resultados, cuya principal responsabilidad recae en el usuario, auxiliado técnicamente por el equipo de desarrollo. El usuario, debe verificar que se satisfacen los requerimientos.

En el caso donde se detecten errores, estos se deben registrar en una bitácora que contenga los siguientes puntos: prioridad; es decir, si la corrección se requiere antes de la aprobación, o bien, antes de la liberación; numerar el error, describir los sintomas de la falla, - quien identificó el error, y fecha del descubrimiento.

En el momento de detectar la falla, se debe proceder a efectuar las correcciones, asignando responsables para depurar el programa o el procedimiento.

Un aspecto importante, que se debe buscar, es probar el sistema hasta sus límites, ya sea en capacidad de - proceso, movimientos efectuados (tanto en la máquina, como operadores humanos), el objetivo de esto es en-- contrar fallas y que el sistema se restablesca en los puntos críticos.

Si después de un esfuerzo, en tiempo y en recursos, el sistema es consistente, y cumple con los criterios de aceptación, el usuario debe dar por aceptado el sistema.

El proceso de pruebas connluye cuando el sistema es usado por el usuario, para producir sus resultados, a esta actividad la denominamos, traspaso del sistema al usuario.

CARACTERISTICAS DE LAS PRUEBAS DE ACEPTACION DEL SISTEMA

- Utilizar el enfoque triple (E)

  Eficaz, eficiente, efectivo para probar el sistema en su totalidad.

- No descuidar los requisitos

  Es decir, en este momento no es recomendable probar a nivel módulo, y además, los cambios detectados en el entrenamiento y conversión, de deben incorporar con anterioridad a esto.

- Esta prueba, debe considerarse como una simulación del trabajo real. Incluyendo casos de tensión como:

  - Apagones simulados

  - Destrucción de archivos.

- Se debe tener, una carga adecuada de datos de prueba y una lista de chequeo de resultados, cuya responsabilidad principal recae en el usuario.

- Llevar una bitácora donde se detecten errores;
  Donde se señalen:

  Prioridad, numeración secuencial del error, descripción de los sintomas de falla, quien indentificó el error, y fecha del sescubrimiento.

- Asignar responsables, para corregir fallas.

- Probar el sistema hasta sus limites.

  (capacidad de proceso, movimientos a la máquina y humanos) restablecimiento en los puntos críticos.

- Aceptar el sistema.

- Traspasar el sistema al usuario.

TECNICAS DÉ PRUEBAS DE ACEPTACION

El conjunto de recomendaciones que hacen eficientes las pruebas, se les denomina: técnicas de pruebas de aceptación, estas consisten en una sistematización de observaciones prácticas, tanto para el equipo de desarrollo como para el usuario.

La primer técnica se denomina, prueba de fallas planeadas; estas consisten en probar los limites de los archivos, y tablas que debe manejar el sistema, así como todo el universo de datos que no se deben aceptar, en los casos de validación y actualización de datos; en el caso de una prueba a un sistema de comunicaciones, se puede recurrir a proporcionar datos confusos o incompletos.

Una segunda técnica son las pruebas de capacidad, estas pretenden descubrir fallas en: los calendarios de producción, o con respecto a la cantidad de terminales -- asignadas para efectuar las transacciones previstas, -- esto se hace para revisar si es el número de estas correspondiente al previsto.

Una tercera *prueba* se denomina *de tensión;* su objetivo consiste en detectar fallas técnicas; tales como, al estar capturando datos, obligar al sistema a que falle, - comprobando que la información previa, exista y no se haya destruído, por ejemplo, apagando la terminal y/o - la máquina.

Finalmente una última técnica de *pruebas*, es la *de respaldo y recuperación*. En esta se buscan, que todos los productos que requieren respaldo, sean en verdad respaldados oportunamente, también, se prevee la recuperación de archivos históricos que sirvan para actualizar, nuevos datos o bien, obtener información para auditoría, dentro de esta técnica, se deben evitar fallas, en la reposición de cintas magneticas, después de un periodo de uso, la restauración de comunicaciones, la recaptura de transacciones omitidas, asegurando que no se duplique la información.

Es importante no descuidar, la revisión de la seguridad de los respaldos, cuando se trate de siniestros o daños provocados por terceros, asegurandose que los respaldos se encuentren bien protegidos.

TECNICAS DE PRUEBAS DE ACEPTACION

- PRUEBAS DE FALLAS PLANEADAS
  - Proporcionar datos excesivos a tablas o archivos.
  - Enviar datos confusos o incompletos a, programas de actualización y/o validación, o de comunicaciones.
- PRUEBAS DE CAPACIDAD
  - Revisión de respuestas oportunas al calendario de producción.
  - Revisión de la cantidad de terminales, con el propósito de conocer, si contamos con el número adecuado de éstas.
- PRUEBAS DE TENSION
  - Obligar al sistema a que se recupere y si falla, mantener intacta la información previa.
  - Esto se puede hacer en el momento de capturar datos, apagando la terminal y/o la máquina.
- PRUEBAS DE RESPALDO Y RECUPERACION
  - Se exámina que todos los productos que requieran respaldo, sean resoaldados oportunamente.
  - Se debe prever, la recuperación de archivos históricos que sirvan para actualizar fúturos datos, o para efecto de presentarlos a los auditores.
  - Cuidar la restauración de cintas.
  - Cuidar la restauración de comunicaciones y recaptura de transacciones.
  - Evitar que se duplique información.
  - Revisión de la seguridad de los respaldos.

CONSIDERACIONES IMPORTANTES EN EL CASO DE PRUEBAS

- No se debe hacer la prueba prematura; es decir, pese a nuestras "urgencias" y las del usuario, se deb previamente elaborar los procedimientos, el entrenamiento y la conversión de lo contrario esto provocaría fallas imprevistas.

- Involucrar al usuario; el riesgo principal de no hacerlo, es quedar encerrado en una torre de marfil, pensando solo en un universo mínimo de pruebas, cuando es el usuario quien mejor conoce el problema real; sin usuario el equipo de desarrollo tendrá doble esfuerzo y tal vez esté perdiendo el tiempo, por otra parte, el usuario debe aceptar el producto, nadie mas puede hacer este trabajo, porque de lo contrario estaría usurpando funciones, es conveniente obtener firmas del usuario toda vez que el producto sea aceptado.

- Manejar adecuadamente los cambios; es necesario incorporar en base a la prioridad cuales cambios deben hacerce en la liberación y cuales deben considerarse como operaciones de mantenimiento, para lo cual se recomienda que esto quede claramente establecido.

- Coordinación precisa de suministros; prever la escasez de paquetes de -- discos, cintas magneticas, y papelería, ya que esto, cuando falta puede provocar actitudes desesperadas, como destruír un archivo histórico o dejar de hacer los respaldos necesarios.

SISTEMAS EN PRODUCCION

- DEFINICION
- REVISIONES
  - CON RESPECTO AL USUARIO
  - DESEMPEÑO DE HARDWARE SOFTWARE
- MEJORAS/CAMBIOS A SISTEMAS EN PRODUCCION
  - DEFINICION
  - IMPACTO
  - RECOMENDACIONES

SISTEMAS EN PRODUCCION

Un sistema está en producción, si los resultados que produce el sistema, son utilizados para satisfacer el conjunto de requerimientos que solicitó el usuario.

En esta fase es recomendable, contar con una aprobación -- por escrito, de la aceptación de pruebas de liberación, -- por parte del usuario; así como, estar seguros que los -- procedimientos y programas se encuentran debidamente actualizados, que el reciclaje de entrenamiento, sea de preferencia responsabilidad del usuario, que la conversión se encuentre completamente terminada y que el sistema se encuentre distribuído tal como fuê previsto.

En esta fase, las revisiones más importantes consisten - en verificar, mediante estadísticas y bitácoras, cual es el desempeño periodico del sistema, existiendo dos clases de revisiones:

- Con respecto al usuario.

  El propósito es revisar los volúmenes y tiempos reales contra los estimados, registrados en las especificaciones.

El tipo de desviaciones que se pueden encontrar son; sobre estimaciones, es decir, inutilización de recursos, tales como terminales o impresoras locales, -- también se puede captar (cambios/mejoras), que incrementen la utilidad el producto.

Finalmente mediante esta revisión se pueden descubrir áreas problema, es decir, lugares que provocan cuellos de botella y por lo tanto afectan la efectividad en el logro de resultados.

- Desempeño del Hardware/software.

Esto consiste en obtener mediciones técnicas, por -- ejemplo, el tiempo de respuesta de un sistema en línea, detectando que una terminal responde en diez segundos en lugar de seis, o bien, en el caso de un - programa de cálculo, en lugar de 45 minutos, su -- tiempo de ejecución es de dos horas, este tipo de observaciones generalmente impacta en el aumento o mejoras en el Hardware/Software.

MEJORAS/CAMBIOS AL SISTEMA EN PRODUCCION

A medida que el usuario ejecuta el sistema, este va detectando mejoras y/o cambios, ya sea que se van acumulando, o bien se van solucionando mediante solicitudes de servicio, solo que estos cambios y/o mejoras, ya no recaen en el equipo de desarrollo, estos son dirigidos al equipo de mantenimiento.

Un equipo de mantenimiento, puede ser organizado por miembros del equipo de desarrollo o por un equipo separado con amplia experiencia en esta clase de actividades.

La administración del equipo de mantenimiento debe detectar la prioridad correcta para satisfacer la mejora/cambio sugerida por el usuario, su responsabilidad también consiste en negociar la posibilidad de hacer o no la mejora/cambio, es decir, debe quedar claro lo urgente de lo necesario, a efecto de proporcionar respuestas oportunas.

Cada vez que son solicitados mejoras/cambios, es recomendable que tanto los usuarios como el equipo de mantenimiento se hagan las siguientes preguntas.

- ¿ El cambio es esencial para alcanzar los objetivos del sistema, originalmente definidos y acordados?

- ¿ Cual va a ser el impacto del cambio propuesto en otras funciones del nuevo sistema, incluyendo las que van a implantar en el futuro y las que ya están en operación?

- ¿ Hasta qué punto, se van a adaptar los usuarios, que tienen poco tiempo de conocer el sistema, a los cambios recientes.

Definitivamente, si los cambios se efectúan al inicio del sistema, estos pueden provocar desconfianza, en los usuarios; se hace dudar de la eficiencia y también se sugiere un inicio de reemplazo del sistema.

Si lo anterior sucede, es un reflejo de que algo en las fases y etapas anteriores no se logró con eficiencia; es conveniente que un sistema no llegue a una alta demanda de mejoras/cambios, esto puede lograrse nejorando técnica y administrativamente todos los procedimientos anteriores; tales como, definición de requerimientos, diseño, construcciín y liberación.

MANTENIMIENTO DE SISTEMAS

- DEFINICION
- FACTORES DELICADOS
- CAUSAS POR LAS QUE EL MANTENIMIENTO ES NECESARIO
- ADMINISTRACION DEL MANTENIMIENTO (RECOMENDACIONES)
- DESARROLLO DEL MANTENIMIENTO

MANTENIMIENTO DE SISTEMAS

A medida que las instalaciones informáticas van liberando sistemas a producción, el mantenimiento cobra más importancia; es un hecho, que cuando todos los sistemas informáticos de una organización se han resuelto en su mayoría, la fase de mantenimiento es la que mas importancia adquiere, esto resulta un reto importante y una extensión vital del proceso de desarrollo de sistemas.

En la actualidad los factores mas delicados en esta etapa son:

- Los costos de mantenimiento son muy altos
- La falta de recursos de programación y de sistemas.
- La falta de un criterio adecuado para que esta etapa sea tratada profesionalmente y no con una actitud negativa.
- La ausencia del uso de metodologías, en la mayoría de los sistemas que se encuentran en esta etapa.
- La falta de documentación actualizada para proporcionar un eficaz y oportuno mantenimiento.
- La falta de recursos financieros, técnicos y operativos, para invertir en un nuevo desarrollo, en un sistema que funciona mal y ha permanecido en mantenimiento durante mucho tiempo.

En la actualidad una manera de minimizar esta clase de problemas; consistiría en ir generando una actitud profesional para ejecutar actividades de mantenimiento; la documentación obligatoria de los sistemas, el uso de metodologías estructuradas, y generadores de programas típicos que sustituyan algunas partes conflictivas del sistema.

El mantenimiento de sistemas, no debe ser visto como un mal necesario; en la actualidad, debemos, en la medida de lo posible de reducirlo; sin embargo, existe un conjunto de causas por las cuales el mantenimiento, no puede ignorarse, y por el contrario, esto provocaría conflictos graves en la organización, algunas de estas son las siguientes:

CAUSAS POR LAS QUE EL MANTENIMIENTO ES NECESARIO.

- Las solicitudes hechas por los usuarios. Pueden incluir la adición de nuevas funciones o nuevas salidas a los sistemas existentes.

- Cambios en las leyes y/o reglamentos de la Administración Pública Federal. Por ejemplo, la Secretaría de Hacienda envía una sustitución de la tabla de impuestos, ò se afecta el tabulador de salarios mínimos.

- Cambios en las operaciones del usuario. Una nueva estructura de precios sustituye a otra existente, se genera una nueva línea de productos, se afectan los criterios para costear productos etc.

- Errores en los sistemas. Como por ejemplo, terminación anormal de programas, incapacidad para ejecutar un trabajo, reportes incorrectos, solicitudes de cambio ignoradas.

ADMINISTRACION DEL MANTENIMIENTO

Por lo anterior expuesto, esta tarea adquiere un grado de interés muy elevado, ya que en este estado del sistema, el administrador debe buscar como uno de sus principales objetivos, disminuír a niveles aceptables los errores en los sistemas.

El administrador debe adelantar, los posibles requerimientos, haciendo las previsiones preventivas en lugar de correctivas.

Para que esto se realice correctamente se requiere que -- exista una asignación adecuada de recursos de programación de calidad, no improvisados y sin experiencia. Recalcamos que debe existir una actitud profesional para este estado del sistema.

Para que un administrador adquiera efectividad en sus actividades, este debe considerar, por lo menos las siguientes recomendaciones.

RECOMENDACIONES PARA ADMINISTRAR EFECTIVAMENTE EL MANTENIMIENTO

- Identificar las solicitudes de mantenimiento

  Se debe detectar si se trata de mejoras/cambios/corrección de errores, en relación a una prioridad; siendo esta urgente, necesaria o recomendable para este tipo de tareas, se recomienda el uso de una bitácora, donde se señale la prioridad así como los recursos.

- Identrificar sistemas problema

  En este sentido, es necesario que el equipo de mantenimiento genere evaluaciones operativas y técnicas, continuas, para evitar problemas.

- Identificar impacto económico

  Este tipo de actividades, son las que elevan la categoría del mantenimiento, ya que el hecho de no efectuarlo, causaría una penalización a la organización. Por lo que se recomienda señalar la importancia en terminos de beneficio/costo cada tarea efectuada por el equipo de mantenimiento.

DESARROLLO DEL MANTENIMIENTO

Dependiendo de la magnitud del mantenimiento, esto puede impactar, desde la actualización de los procedimientos y controles del usuario, hasta el desarrollo total de programas; ya sea, que aumenten las funciones o que se sustituyan programas, recordando que si de esto se trata, es recomendable utilizar los estandares y metodologías estructuradas, utilizadas por los desarrolladores de sistemas.

El conjunto de etapas mínimas que se deben realizar en el mantenimiento son.

- Identificar el probelma y sus causas

  La forma mas facil es conociendo del mismo usuario y como se presenta el problema.

- Plantear una solución

  Esto implica plantear, un método de solución, el cual puede implicar, el rediseño de procedimiento o el diseño de programas/módulos.

- Realizar la construcción de programas/módulos

  Esto requiere básicamente de codificar y probar los programas/módulos generados.

- Efectuar revisiones

  Básicamente se busca la aprobación del usuario, es decir que a través de los datos de prueba que proporcione y en base a una lista de resultados esperados, el programa/módulo solucione el problema.

- Instalar la solución

  Reemplazar o adicionar los programas/módulos, - que solucionan el problema, y actualizar previamente los manuales de procedimientos.

El mantenimiento de sistemas, existe y continuará existiendo por mucho tiempo, y solo para que esto funcione realmente, se requiere de una actitud profesional.

Es necesario, que los equipos de mantenimiento hagan uso de las técnicas necesarias para cumplir correctamente con su tarea, esto se puede lograr construyendo herramientas que mejoren la calidad y el esfuerzo de mantenimiento, en

el menor tiempo posible.

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

ANEXOS

FEBRERO, 1984

FASES DE PLANEACION

DEFINICION

LA PLANEACIÓN ES PROYECTAR UN FUTURO DESEADO Y LOS MEDIOS EFECTIVOS PARA CONSEGUIRLO.

OBJETIVO

EL OBJETIVO BÁSICO DE LA FASE DE PLANEACIÓN ES ELABORAR UN PLAN PARA LA ADMINISTRACIÓN Y EL DESARROLLO TÉCNICO DEL PROYECTO.

PARTES DE LA PLANEACION

DICHAS PARTES NO REPRESENTAN EL ORDEN ESTRICTO EN EL QUE SE DEBEN LLEVAR A CABO.

FINES

ESPECIFICAR METAS Y OBJETIVOS.

MEDIOS

ELEGIR POLÍTICAS, PROGRAMAS, PROCEDIMIENTOS Y PRÁCTICAS CON LAS QUE HABRÁ DE ALCANZARSE LOS OBJETIVOS.

RECURSOS

DETERMINAR TIPOS Y CANTIDADES DE LOS RECURSOS QUE SE NECESITAN,DEFINIR COMO SE HABRÁN DE ADQUIRIR O GENERAR, Y COMO HABRÁN DE ASIGNARSE A LAS ACTIVIDADES.

REALIZACION

DISEÑAR LOS PROCEDIMIENTOS PARA TOMAR DECISIONES, ASÍ COMO LA FORMA DE ORGANIZARLOS PARA QUE EL PLAN PUEDA REALIZARSE.

CONTROL

DISEÑAR UN PROCEDIMIENTO PARA PROVEER O DETECTAR LOS ERRORES O LAS FALLAS DEL PLAN, ASÍ COMO PARA PREVENIRLOS O CORREGIRLOS SOBRE UNA BASE DE CONTINUIDAD.

CARACTERISTICAS DE LA PLANEACION TACTICA Y ESTRATEGICA

	PLANEACIÓN TÁCTICA	PLANEACIÓN ESTRATÉGICA
EFECTOS DE UN PLAN	MÁS CORTOS Y REVERSIBLES	MÁS LARGOS, DURADEROS E IRREVERSIBLES.
INFLUENCIAS EN LAS COMPONENTES DE LA ORGANIZACIÓN.	MENORES Y DE PERSPECTIVAS MÁS LIMITADAS	MAYORES Y DE PERSPECTIVAS MÁS AMPLIAS
HORIZONTE DE PLANEACIÓN	MÁS A CORTO PLAZO, I.E. MÁS EFICIENTE EN UN HORIZONTE DE PLANEACIÓN MÁS LIMITADO.	MÁS A LARGO PLAZO
OBJETIVOS	CONSIDERA MEDIOS, FINES PRINCIPALMENTE	CONSIDERA METAS Y FINES

CONCLUSIÓN: LA DISTINCIÓN ENTRE ESTOS DOS TIPOS DE PLANEACIÓN ES MÁS RELATIVA QUE ABSOLUTA.

EN UN PLAN SE NECESITAN AMBAS PARA OBTENER EL MÁXIMO BENEFICIO.

ACTIVIDADES

- DEFINICIÓN DE LOS REQUERIMIENTOS TÉCNICOS DEL SISTEMA

- DESCRIPCIÓN DE LA DOCUMENTACIÓN

- LOS PRODUCTOS A PRODUCIR

- LAS ACTIVIDADES Y REVISIONES A REALIZAR

- PREPARACIÓN DE ESTÁNDARES Y NORMAS

- DEFINICIÓN DEL MECANISMO DE CONTROL

- ESTABLECER LOS REQUERIMIENTOS DE UNA "BIBLIOTECA DE SOPORTE DE PROGRAMAS".

PRODUCTOS FINALES DE LA FASE

- MATRIZ DE REQUERIMIENTOS

- RUTA CRÍTICA DE ACTIVIDADES

- ESTABLECIMIENTO DE RECURSOS

- NORMAS Y ESTÁNDARES DE DOCUMENTACIÓN.

ESTUDIO DE FACTIBILIDAD

- OBJETIVOS DEL ESTUDIO DE FACTIBILIDAD.

- INTEGRACIÓN DE LOS GRUPOS DE TRABAJO

    - GRUPO COORDINADOR

    - GRUPO TECNICO

- ELABORACIÓN DEL PLAN DE TRABAJO

- PRESENTACIÓN DEL ESTUDIO DE FACTIBILIDAD.

- CONSIDERACIONES CLAVES.

- EVALUACIÓN ECONÓMICA DEL PROYECTO.

OBJETIVO DEL ESTUDIO DE FACTIBILIDAD [8]

EL OBJETIVO DEL ESTUDIO DE FACTIBILIDAD, ES INFORMAR A LA ADMINISTRACIÓN DE LA EMPRESA, LAS CARACTERÍSTICAS PROBABLES, COSTOS Y BENEFICIOS AL LLEVAR A LA PRÁCTICA UN SISTEMA ESPECÍFICO Y PROPORCIONAR LOS REQUERIMIENTOS GENERALES REFERENTES AL PROYECTO EN FORMA DE REPORTE MANUAL..

INTEGRACION DE LOS GRUPOS DE TRABAJO

LA REALIZACIÓN DEL ESTUDIO REQUIERE DE LA FORMACIÓN O CONSTITUCIÓN DE DOS GRUPOS, UNO DE LOS CUALES DEBERÁ FUNGIR COMO COORDINADOR DEL ESTUDIO Y EL SEGUNDO COMO EJECUTOR DEL MISMO.

GRUPO COORDINADOR

El Grupo Coordinador hace participar a los Directores de la Empresa en las decisiones que habrán que tomarse a lo largo del estudio.

FUNCIONES

- Definir los objetivos específicos y la cobertura del estudio.

- Integración del grupo técnico que se encargará del desarrollo.

- Proveer al grupo técnico de los elementos de apoyo necesarios para su correcta operación.

- Servir como enlace entre las áreas involucradas en el estudio y el grupo técnico.

- Someter a consideración de los Directores de la Empresa los objetivos del estudio.

- Dirigir y controlar permanentemente el desarrollo del estudio.

- Analizar y evaluar los resultados finales.

GRUPO TECNICO

El Grupo Técnico estará formado por especialistas en: Análisis de sistemas, Procesamiento de Datos, Investigación de requerimientos de información, Métodos e instrumentos de capacitación, técnicas en diseminación y flujos de información, etc.

FUNCIONES

- Definir las tácticas de acción para el desarrollo.
- Elaborar el plan de trabajo y programa de actividades.
- Someter los puntos anteriores para su aprobación.
- Desarrollar y documentar las diferentes etapas del estudio.
- Realizar la integración y síntesis del estudio.
- Llevar a cabo la selección y presentación de alternativas.
- Implementar las soluciones adoptadas por el grupo coordinador.

ELABORACION DEL PLAN DE TRABAJO (12)

EL ADMINISTRADOR DEL PROYECTO DEBE RECONOCER QUE LOS PLANES DE TRABAJO REQUIEREN DE UNA ACTUALIZACIÓN Y VERIFICACIÓN CONTÍNUAS.

EL PLAN DE TRABAJO DEBE DE CONTENER, ENTRE OTROS.

- NÚMERO DE OPERACIONES ACTUALES QUE SE REVISARÁN

- DESIGNAR A LOS ENTREVISTADOS Y ENTREVISTADORES.

- LOS TEMAS

- LA DURACIÓN PLANEADA PARA CADA ENTREVISTA.

- EL TRABAJO CALCULADO PARA ELABORAR LA DOCUMENTACIÓN.

- FECHAS ESPECÍFICAS DE INICIO Y TERMINACIÓN.

PRESENTACION DEL ESTUDIO DE FACTIBILIDAD

El Reporte Ejecutivo deberá incluir por lo menos una representación clara y concisa de:

- Objetivos del Estudio
- El campo de acción
- El resúmen de las consideraciones económicas
- Posibles alternativas

Al concluir el Estudio de Factibilidad, el Director de la Empresa está en posición de aceptar o rechazar las recomendaciones del Equipo del Proyecto.

CONSIDERACIONES CLAVES

14

- APROBACIÓN FORMAL DEL TRABAJO EFECTUADO DURANTE EL ESTUDIO DE FACTIBILIDAD.

- COMPROMISO FORMAL SOBRE LOS RECURSOS DISPONIBLES, POR PARTE DE LOS USUARIOS Y LA ADMINISTRACIÓN GENERAL.

- DEFINIR LA PRIORIDAD DEL PROYECTO

EVALUACION ECONOMICA DEL PROYECTO

LA FINALIDAD ES COMPARAR EL DESARROLLO ESTIMADO Y LOS COSTOS OPERATIVOS CON LOS BENEFICIOS.

LA EVALUACIÓN ECONÓMICA DEBE INCLUIR:

- LOS COSTOS DE DESARROLLO
- LOS COSTOS OPERATIVOS CUANTIFICABLES
- OTROS COSTOS
- BENEFICIOS TANGIBLES
- BENEFICIOS INTANGIBLES
- HIPÓTESIS Y LIMITACIONES OPERATIVAS

<u>La estimación de costos en el Desarrollo de Sistemas.</u>

- Objetivos y necesidades
- Diversos factores y problemas
- Técnicas de estimación
  - Método de estimación de tiempo de programación
  - Método de Eriksen
  - Método de Myers
  - Estadísticas de Walston y Felix
  - Método de Putnam

OBJETIVOS Y NECESIDADES DE LA ESTIMACION DE COSTOS

- AYUDAR A LA TOMA DE DECISIONES SOBRE LA OPORTUNIDAD DE APLICACIONES Y SOBRE LAS DIVERSAS ALTERNATIVAS QUE SE PLANTEAN.

- AYUDAR A LA GESTIÓN DEL PROYECTO INFORMÁTICO.

- AYUDAR A LA EVALUACIÓN DEL PERSONAL.

DIVERSOS FACTORES Y PROBLEMAS

- El plazo
    - Actividades secuenciales
    - Cuellos de botella
    - Aprendizaje y la intercomunicación
- La calidad
    - Eficacia
    - Eficiencia
    - Adecuación a estandares
    - Ausencia de errores en los programas
- Los métodos
    - Herramientas automaticas
    - Complejidad de las herramientas
- Los recursos disponibles
    - Humanos
    - Materiales

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

ADMINISTRACION DE PROYECTOS EN INFORMATICA

REFERENCIAS

FEBRERO, 1984

REFERENCIAS

1. ADMINISTRACION DE PROYECTOS DE SOFTWARE
PORTILLA ROBERTSON
DECFI, UNAM
1982

2. AUTOMATED TOOLS FOR INFORMATION SYSTEMS DESIGN
H. J. SCHNEIDER AND A.I. WASSERMAN
NORTH-HOLLAND
1982

3. CLASSICS IN SOFTWARE ENGINEERING
EDWARD NASH YOURDON
YOURDON PRES

4. COMPOSITE/STRUCTURED DESIGN
GLENFORD J. MYERS
VAN NOSTRAND REINHOLD CO.
1978

5. FORMAL MODELS AND PRACTICAL TOOLS FOR INFORMATION SYSTEMS DESIGN
HANS-JOCHEN SCHNEIDER
NORTH HOLLAND
1979

6. GUIA PARA EL DESARROLLO DE PLANES DE PRUEBA
DR. RONALDO NIEVA GOMEZ
I.I.E.
1982

7. MANAGING THE STRUCTURED. TECHNIQUES
EDWARD YOURDON
PRENTICE-HALL
1979

8. SOFTWARE ENGINEERING
JENSEN
PRENTICE-HALL
1979

9. SOFTWARE ENGINEERING. A PRACTITIONER'S APPROACH
ROGERS PRESSMAN, P.H.D.
McGRAW-HILL
1982

10. SOFTWARE ENGINEERING DESIGN, RELIABILITY AND MANAGEMENT
MARTIN L. SHOOMAN
McGRAW-HILL
1983

11. SOFTWARE ENGINEERING ECONOMICS
BARRY W. BOEHM
PRENTICE-HALL
1982

12. SOFTWARE LIFE CYCLE (HP-SLC)
HEWLETT PACKARD CO.
1983

13. SOFTWARE TESTING TECHNIQUES
BORIS BEIZER
VAN NOSTRAND REINHOLD CO.
1983

14. STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION
TOM DEMARCO
YOURDON INC.
1978

15. STRUCTURED SYSTEMS ANALYSIS TOOLS AND TECHNIQUES
CHRIS GANE AND TRISH SARSON
PRENTICE-HALL
1979

16. STRUCTURED SYSTEMS DEVELOPMENT
KENETH T. ORR
YOURDON PRESS
1977

17. STRUCTURED METHODOLOGY FOR SOFTWARE DESIGN AND DEVELOPMENT
STATE OF THE ART SEMINARS
1981

18. SYSTEM DEVELOPMENT METHODOLOGY
G.F. HICE
W.S. TURNER, Ph.D.
L.F. CASHWELL
NORTH-HOLLAND PUBLISHING
1974

19. TECNICAS MODERNAS DE DESARROLLO Y ADMINISTRACION DE PROGRAMACION
I.I.E.
DECFI, UNAM
1983

20. THE ART OF SOFTWARE TESTING
GLENFORD J. MYERS
ED. EL ATENEO
1979

21. THE MYTHICAL MAN MONTH
BROOKS, F.P. Jr.
ADDISON-WESLEY
1975

22. PROGRAM DESIGN AND CONSTRUCTION
HIGGINS DAVID A.
ENGLEWOOD CLIFFS; PRENTICE-HALL
1979

23. PRINCIPLES OF PROGRAM DESIGN
JACKSON M.A.
NEW YORK: ACADEMIC PRESS
1975

24. STRUCTURED REQUIREMENTS DEFINITION
ORR, KENNETH T.
TOPEKA: KENORR & ASSOCIATES, INC.
1980

25. LOGICAL CONSTRUCTION OF PROGRAMS
WARNIER JEAN-DOMINIQUE
NEW YORK: VAN NOSTRAND REINHOLD, CO.
1976

26. LOGICAL CONSTRUCTION OF SYSTEMS
WARNIER JEAN-DOMINIQUE
NEW YORK: VAN NOSTRAND REINHOLD, CO.
1981

27. PROGRAM DESIGN
BLAISE W. LIFFICK, EDITOR
NEW HAMPSHIRE: BYTE PUBLICATIONS, INC.
1978

28. DESIGNING STRUCTURED PROGRAMS
HIGGINS DAVID A.
ENGLEWOOD CLIFFS; PRENTICE-HALL
1983

29. PRACTICAL LCP
GARDNER ALBERT, C.
ENGLAND: MACGRAW HILL BOOK, CO.
1981

30. DATA BASE DESIGN METHODOLOGY
M. VETTER Y R.N. MADDISON
ENGLEWOOD CLIFFS: PRENTICE-HALL INTERNATIONAL
1981

31. SYSTEM DEVELOPMENT
JACKSON MICHAEL
ENGLEWOOD CLIFFS: PRENTICE-HALL INTERNATIONAL
1983

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

ADMINISTRACION DE PROYECTON EN INFORMATICA

EJERCICIO

FEBRERO, 1984

1ER METODO

EJEMPLO

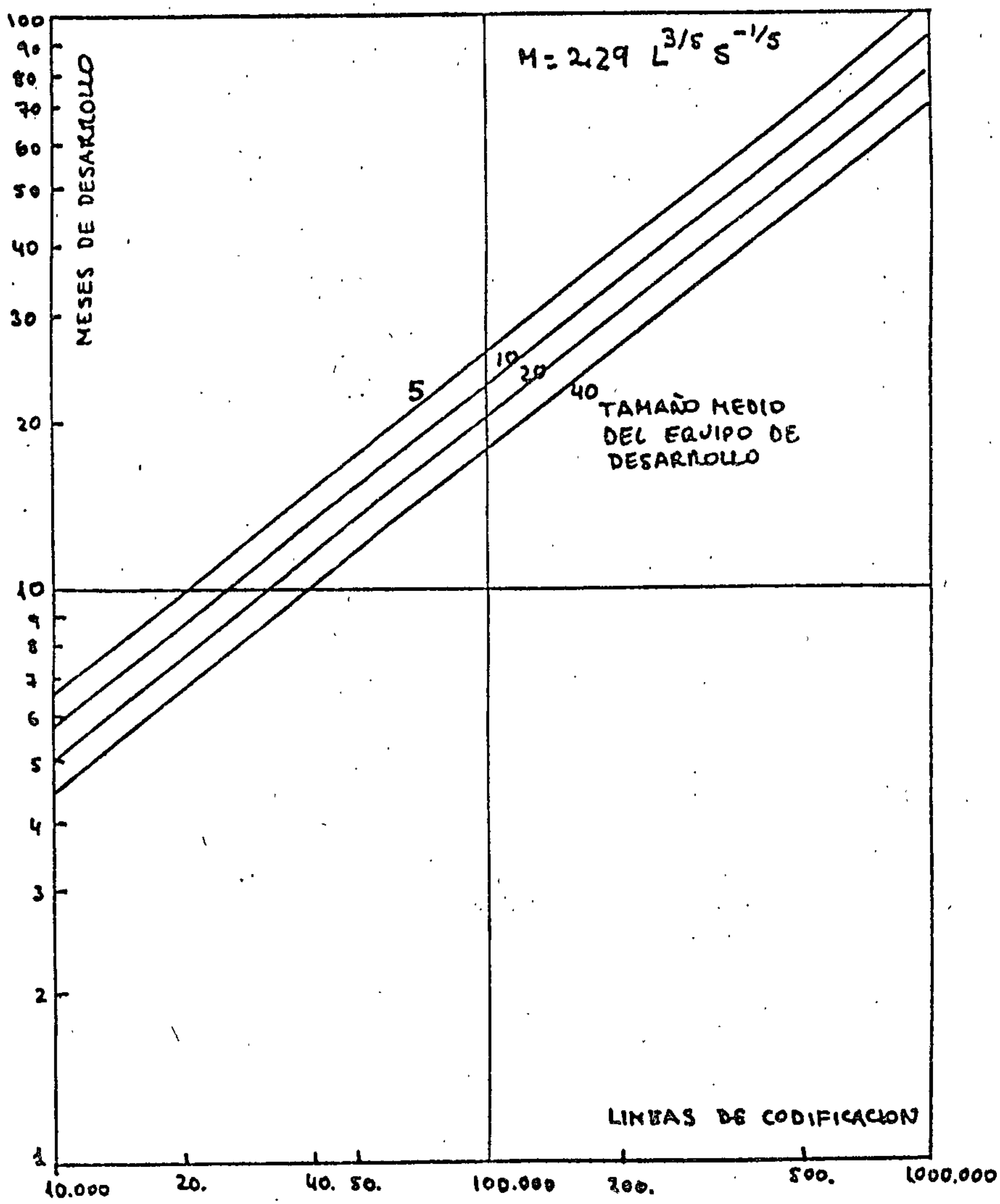
PROGRAMA QUE LEE UN FICHERO SECUENCIAL (VARIOS TIPOS DE REGISTRO) Y ACCEDE A UN FICHERO IS, OBTENIENDO UN FICHERO SECUENCIAL Y UN LISTADO CON LO TIPOS DE LINEAS. SON RELATIVAMENTE COMPLEJAS LAS FUNCIONES DE REESTRUCTURACION Y PRESENTACION DE DATOS, Y SIMPLES EL RESTO. VA A SER REALIZ. POR UN PROGRAMADOR EXPERIMENTADO, EN COBOL

C : ENTRADAS Y SALIDAS		
- FICHERO SECUENCIAL ENTRADA	2	
- FICHERO IS	3	
- FICHERO SECUENCIAL SALIDA	2	
- LINEAS DE IMPRESION	10	
	17	
PROCESOS		
- DATOS	3	
- CONDICIONES	1	
- PRESENTACION	5	
- CALCULO	3	
- encadenamiento	1	
	13	
C : TOTAL		30

A : 1

B : 0 (SE SUPONE QUE NO SE NECESITA CONOCIMIENTO DEL PROBLEMA)

E = 30 (1+0) = 30 DIAS-HOMBRE


M = 2.29 $L^{3/5}$ $S^{-1/5}$
MESES DE DESARROLLO
100
90
80
70
60
50
40
30
20
10
9
8
7
6
5
4
3
2
1
5
10
20
40
TAMAÑO MEDIO DEL EQUIPO DE DESARROLLO
LINEAS DE CODIFICACION
10.000
20.
40.
50.
100.000
200.
500.
1000.000

PUTNAM

EJEMPLO

APLICACION DE UNAS 100.000 LINEAS FUENTE. COSTE POR AÑO-HOMBRE TOTAL = 4 MPTS.

a) LA = 160.000 LM = 90.000 LB = 80.000

$$L = \frac{80 + 4 \times 90 + 160}{6} = 100.000$$

$$\sigma = \frac{160 - 80}{6} = 13.000$$

RIESGO <	50%	16%	3%
DIMENSION	100.000	113.000	126.000

b) CALCULO PARA VARIOS TAMAÑOS DE EQUIPO MEDIO

		MESES			COSTE (M)			PRODUCTIVIDAD (L		
S	R	50	16	3	50	16	3	50	16	3
5		26	28	30	48	52	55	48	50	52
10		23	25	26	84	90	96	27	28	30
20		20	21	23	146	157	168	16	17	17

- SI SE DESEA ACABAR EN 23 MESES CON PROBABILIDAD MAYOR DEL 97% : S = 20 , C = 168 Mpts P = 17.
- SI SE ADMITE PROB 84% : S = 10 C = 84 Mpts P = 2

c) PREVISION DE COSTE DE MANTENIMIENTO:

1,5 × COSTE DEL PROYECTO DE DESARROLLO

WALSTON

EJEMPLO

PROYECTO DE 100.000 LINEAS, CON INDICE DE PRODUCTIVIDAD O :

P = (469 , 274 , 160)

EA (PESIMISTA, 1σ)	625	M-H
EM	365	M-H
EB (OPTIMISTA, 1σ)	213	M-H

D (13.500 , 4.900 , 1.800) PAGINAS

M (38,3 , 21,5 , 12,1) MESES

S (PARA 365 M-H) (29 , 19 , 12) PERSONAS

C (369 , 153 , 63) MILES DE DOLARES DE EQUIPO

ERIKSEN

EJEMPLO

APLICACION DE COMPLETIDAD MEDIA-FACIL, CON UNAS 100.000 LINEAS PREVISTAS, EN COBOL

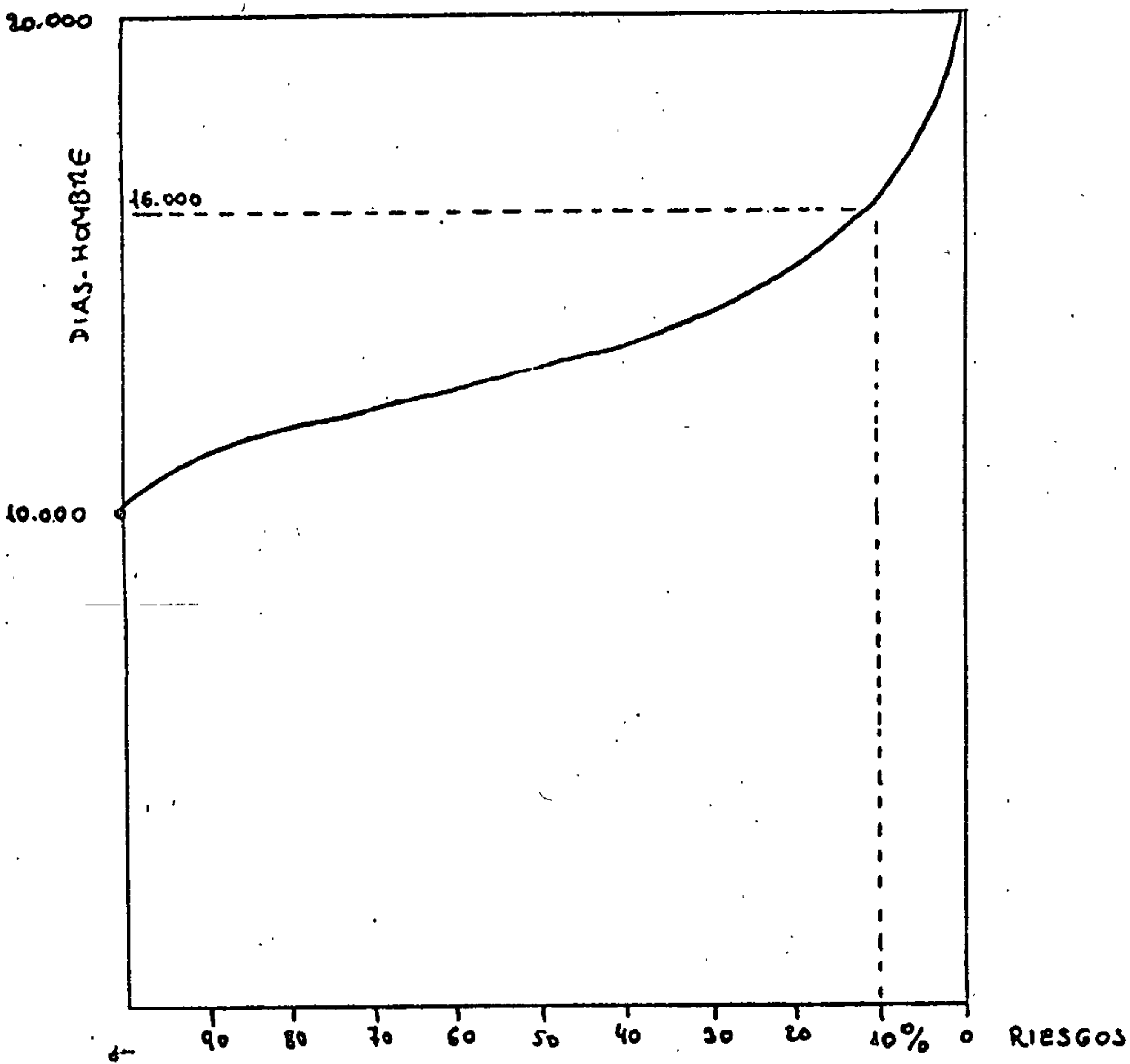
PROGRAMAS

TIPO	TAMAÑO	COMPLEJIDAD	NUMERO	LINEAS	D.H	H.M
E	(10)		50	~ 500	200	-
N	1 (200)	A	20	4,000	280	60
N	2 (500)	B	20	10.000	360	100
N	2 (500)	C	20	10.000	820	120
N	3 (700)	B	20	14.000	700	120
N	3 (700)	C	20	14.000	900	210
N	4 (900)	B	10	9.000	390	70
N	4 (900)	C	9	8.100	441	72
N	4 (900)	D	1	900	60	10
N	5 (1100)	B	5	5.500	215	40
N	5 (1100)	C	5	5.500	265	45
N	5 (1100)	D	5	5.500	320	55
N	6 (1300)	C	5	6.500	285	50
N	6 (1300)	D	5	6.500	340	60
TOTAL			195	100,000	5,836	1,012

TOTAL PROYECTO : D.H : 2,5 × 5836 = 14,590 ≃ 912 M-H

H.M : 1,012

NO TIENE EN CUENTA PLAZO (O TAMAÑO DE GRUPO)
PROBABILIDADES


20.000
16.000
DIAS-HOMBRE
10.000
90
80
70
60
50
40
30
20
10%
0
RIESGOS

DIRECTORIO DE LOS ALUMNOS DEL CURSO DE ADMINISTRACION DE PROYECTOS EN INFORMATICA (DEL 13 AL 24 DE FEBRERO DE 1984)

NOMBRE Y DIRECCION — EMPRESA Y DIRECCION

1. DANIEL ALCARAZ MARTELL
Rancho "El Encanto" No. 24
Col. Fracc. Sta. Cecilia
Deleg. Coyoacán
México, D. F.
594-8002

DIRECCION GRAL. DE INFORMATICA DE INGRESOS SHCP
Izazaga No. 89 7° Piso
Col. Centro
Deleg. Cuauhtémoc
México, D. F.

2. GABRIEL ADRIAN ALVAREZ ORTEGA
Luz Saviñón No. 623-4
Col. Del Valle
Deleg. Benito Juárez
México, D. F.
687-0222

BANAMEX
Fray Servando T. de Mier No. 42-11
Col. Centro
Deleg. Cuauhtémoc
06000 México, D. F.
761-7319

3. GUILLERMO AMEZQUITA SAHAGUN
Dr. Barragán No. 780-6
Col. Narvarte
Deleg. Benito Juárez
03020 México, D. F.
696-5482

BANCO DE MEXICO - FIRA
Av. Insurgentes Sur No. 2375
Col. Sn Angel
01080 México, D. F.
550-9432

4. JUAN ARMAS MENESES
Aguascalientes No. 99-1
Col. Roma
Deleg. Cuauhtémoc
México, D. F.

IMPORTADORA Y EMP. DE A. Y DER. S.A.
Calz. Ignacio Zaragoza No. 1160
Col. Pantitlán
Deleg. Iztacalco
México, D. F.

5. SEMIRAMIS ANGELES MUÑOZ
Edif 36-A
Col. Nueva Sta. María
Deleg. Azcapotzalco
02800 México, D. F.
556-4792

PETROLEOS MEXICANOS
Ejército Nacional No. 418
México, D. F.
244-14

6. ANTONIO ARANDA PASTOR
Sánchez Azcona No. 1651-801
Col. Del Valle
Deleg. Benito Juárez
03100 México, D. F.
688-6848

PETROLEOS MEXICANOS
Ejército Nacional No. 436 9° Piso
Col. Chapultepec Morales
México, D. F.
545-1455

7. MIGUEL ANGEL ARTEAGA CASAS
Edif. 251 Ent "B" Depto 204
Col. Acueducto Guadalupe
Deleg. Gustavo A. Madero
07270 México, D. F.
392-3099

SECRETARIA DE EDUCACION PUBLICA
Argentina No. 28 Puerta 210
Col. Centro
Deleg. Cuauhtémoc
México, D. F.
521-5561

... 2

8. LUIS ANTONIO ARZUBIDE ARZUBIDE
Emiliano Zapata No. 50 Int.101
Col. Centro
Deleg. Cuauhtémoc
06060 México, D. F.
542-3264

BANCO NACIONAL DE MEXICO, (BANAMEX)
Fray Servanco T. de Mier No. 42-11
Col. Centro
Deleg. Cuauhtémoc
México, D. F.
588-7284

9. JOSE ALBERTO CABRERA LOPEZ
Ernesto M. Fierro No. 121
Col. Fracc. Colonial
Deleg. Iztapalapa
09270 México, D. F.
691-7554

S.C.T.
Xola y Av. Universidad
Col. Narvarte
México, D. F.
530-1198

10. RAUL FELIPE CORTES CORONADO
Ma. Contreras No. 36
Col. San Rafael
Deleg. Cuauhtémoc
06470 México, D. F.
535-8770

FONDO DEL PROGRAMA DESCENTRALIZACION
DE LAS EXPLORACIONES LECHERAS DEL D.F.
Hamburgo No. 31
Col. Juárez
Deleg. Cuauhtémoc
06600 México, D. F.
566-5438

11. OSCAR COVARRUBIAS AGUIRRE
Puente No. 87
Col. Jardínes del Sur
Deleg. Xochimilco
16050 México, D. F.
676-0518

S.E.D.U.E.
Universidad frente a Mítla s/n
Col. Narvarte
Deleg. Benito Juárez
03020 México, D. F.
590-4105

12. JAIME CRISTO ALVAREZ
J.J. Fernández de Lizardi 146
Col. Cd. Satélite
53100 Estado de México
562-2369

AUTOTRANSPORTES TRES ESTRELLAS DE ORO
Calz. Vallejo No. 1268
Col. Santa Rosa
México, D. F.
391-1109

13. PABLO E. CRUZ AGREDA
Mar Blanco No. 2
Col. Fracc. Lomas Lindas
Deleg. Atizapán,
54500 Estado de México

INSTITUTO MEXICANO DE COMERCIO EXT.
Alfonso Reyes No. 30
Col. Condesa
Deleg. Cuauhtémoc
06140 México, D. F.
211-0036 ext 132

14. EDUARDO DOMINGUEZ ESPINOSA
Navarra No. 250-2
Col. Alamos
Deleg. Benito Juárez
00340 México, D. F.
696-1266

SECRETARIA DE DESARROLLO URBANO Y
ECOLOGIA
Av. Universidad frente a Mítla
Col. Narvarte
Deleg. Benito Juárez
03020 México, D. F.
590-6424

15. HECTOR HUGO DOMINGUEZ RICO
Capuchinas No. 53 -104
Col. La Concordia
Deleg. Lomas Verdes
12053 México, D. F.

GENERAL MOTORS DE MEXICO, S. A. DE C.V.
Lavo Victoria No. 74
Col. Granada
Deleg. Miguel Hidalgo
México, D. F.
254-3777 ext. 204

... 3

16. JOSE LEOBARDO FENTANES MTZ.
Lago Victoria No. 74
Col. Nueva Granada
Deleg. Miguel Hidalgo
México, D. F.
254-3777

GENERAL MOTORS DE MEXICO, S.A. DE C.V.
Lago Victoria No. 74
Col. Nueva Granada
Del. Hidalgo
México, D. F.
244-3777

17. IGNACIO JAIME FERRER OCHOA
F. C. Interoceanico No. 69
Col. Morelos
Deleg. V. Carranza
México, D. F.

CASA DE BOLSA MADERO, S.A.
Reforma No. 90
Col. Juárez
Deleg. Cuauhtémoc
06600 México, D. F.
592-5599 ext. 281

18. JUAN CLAUDIO FLORES CALDERON
Paseo de los Pensamientos 8
Col. Hda. Ojo de Agua Tecamal
55770 Edo. de México
91 595 80363

GENERAL MOTORS DE MEXICO, S.A. DE C.V.
Lago Victoria No. 74
Col. Hidalgo
México, D. F.
254-3777 ext. 406

19. CUAUHETEMOC A. FUENTES BUCIO
Las Flores No. 35-304
Col. Coyoacán
Deleg. Coyoacán
04380 México, D. F.
544-7683

CASA DE BOLSA MADERO, S.A.
Reforma No. 90 3° Piso
Col. Juárez
Deleg. Cuauhtémoc
06600 México, D. F.
592-2666 592-5599

20. NICOLAS RAMON GALLEGOS ARTEAGA
Gpo. 9-40 Uss No. 2
Tlanepantla
54030 Edo. de México
565-1360

S.C.T. DIR. GRAL. AUTROTRANSPORTE FED.
Calz de las Bombas No. 411
Col. Villa Coapa
Deleg. Coyoacán
México, D. F.
684-4420

21. JOSE LUIS GARCIA ANGULO
11 de Abril No. 177-403 B
Col. Sn. Pedro de los Pinos
Deleg. Benito Juárez
01311 México, D. F.
271-4651

DIRECCION GRAL. DE AEROPUERTOS S.C.T.
Chiapas No. 121 P.B.
Col. Roma
Deleg. Cuauhtémoc
06760 México, D. F.
574-8340

22. LEONARDO GARCIA LOPEZ
Calz. San Simón No. 242
Col. Atlampa
Deleg. Cuauhtémoc
06450 México, D. F.
541-4531

23. JOSE LUIS GARCIA OLIVERA
Gpe. I. Ramírez No. 64
Col. Tepepan
Deleg. Xochimilco
16020 México, D. F.
676-1962

DIRECCION GRAL. DE OBRAS MARITIMAS
Insurgentes Sur No. 664
Col. Del Valle
13100 México, D. F.

... 4

24. MA. DE LA LUZ GOMEZ ROJAS
Manuel Doblán No. 29-202 A
Col. Tacubaya
Deleg. Miguel Hidalgo
11870 México, D. F.

PROCURADURIA FEDERAL DEL CONSUMIDOR
Dr. Carmona y Valle No. 11
Col. Doctores
Deleg. Cuauhtémoc
México, D. F.
761-3811 ext. 214

25. ARMANDO JASSO ARIAS
Calle 3 No. 152
Col. San Pedro de los Pinos
México, D. F.

SRIA.DE AGRICULTURA Y RECURSOS HIDRAULICOS
Reforma No. 107 8° Piso
Deleg. Cuauhtémoc
México, D. F.
592-1062

26. CARLOS ALBERTO LOBOS SOTO
Londres No. 190 Depto. 214
Col. Juárez
Deleg. Cuauhtémoc
06600 México, D. F.
514-3756

I.C.A. (CONSTRUCTORES METRO, S.A. DE C.V.)
Altadena No. 23 9° Piso
Col. Nápoles
Deleg. Juárez
México, D. F.
687-2371

27. VICTOR MENDIZABAL LOPEZ
Ing. José J. Reynoso No. 68
Col. Constitución de 1917
Deleg. Iztapalapa
09260 México, D. F.
691-3779

S.A.R.H.
Insurgentes Sur No. 30
Col. Benito Juárez
Deleg. Cuauhtémoc
México, D. F.
591-1835

28. ARMANDO LOPEZ ROMERO
Las Flores No. 25
Col. Los Reyes
Deleg. Coyoacán
04330 México, D. F.
549-7479

CENTRO DE INFORMACION CIENTIFICA Y HUMANISTICA
Ciudad Universitaria
México, D. F.
550-5215 ext. 4212

29. MARIO ALEJANDRO LOPEZ TRENADO
2° Cda. Calle 10 No. 224-4
Col.Iztapalapa
09070 México, D. F.
581-3812

SECRETARIA DE EDUCACION PUBLICA
Argentina No. 28
Col. Centro
Deleg. Cuauhtémoc
México, D. F.
521-5561

30. FRANCISCO JAVIER MTZ. CAVAZOS
Cerrada del Pino No. 17
Col. R. Valle Dorado
Deleg. Tlalnepantla
54020 México, D. F.
370-0017

PROMOCIONES INDUSTRIALES MEXICANAS S.A. DE C. V.
Paseo de las Palmas No. 755 8° Piso
Col. Lomas de Barrilaco
Deleg. Miguel Hidalgo
11010 México, D. F.
540-7650

31. JOSE LUIS MARTINEZ CERVANTES
Tibio Muñoz No. 31
Col. Olímpica
Deleg. Naucalpán
Estado de México

GENERAL MOTORS DE MEXICO, S.A. DE C.V.
Lago Victoria No. 74
Col. Ampl. Granada
Deleg. Miguel Hidalgo
México, D. F.
254-3777

... 5

32. ABEL MARTINEZ MARQUEZ
Lago Victoria No. 74
Col. Ampliación Granada
Deleg. Hidalgo
México, D. F.

GENERAL MOTORS DE MEXICO, S.A. DE C.V.
Lag. Victoria No. 74
Col. Ampliación Granada
Deleg. Hidalgo
México, D. F.
254-7777

33. ALFONSO MARTINEZ REYES
Fray Servando T. de Mier 112-2
Col. Centro
Deleg. Cuauhtémoc
06080 México, D. F.
761-1518

DIRECCION GRAL. DE AUTROTRANSPORTE FED
Calz. de las Bombas No. 411 - mezzanin
Col. San Bartolo Coapa
México, D. F.
684-4664

34. DANIEL MARTINEZ TAMARIZ
Lago Victoria No. 74
Col. Nueva Granada
Deleg. Hidalgo
México, D. F.

GENERAL MOTORS
Lago Victoria No. 74
Col. Nueva Granada
Deleg. Hidalgo
México, D. F.
254-3777

35. ANA MARIA MARTINEZ_ TORRES
Ahuehuetes No. 9
Col. Iztacalco
Deleg. Iztacalco
08900 México, D. F.
650-1079

BANCO SOFIMEX, S.N.C.
Bolivar No. 18 - 3° Piso
Col. Centro
Deleg. Cuauhtémoc
06000 México, D. F.
585-2133 ext 152 164

36. DELFINO MARTINEZ TORRES
Calle Lucio Blanco No. 60
Col. Providencia
Deleg. Atzcapotzalco
02440 México, D. F.
352-0451

DIRECCION GRAL. DE AEROPUERTOS, S.C.T
Chiapas No. 21
Deleg. Cuauhtémoc
México, D. F.
574-8892

37. ANA MARIA MARTINEZ

BANCO SOFIMEX, S.N.C.
Bolivar No. 8 Desp. 208
Col. Centro
Deleg. Cuauhtémoc
06000 México, D. F.
585-3954

38. GABRIEL MONTES HERNANDEZ
Oriente No. 243-113
Col. Agrícola Oriental
Deleg. Iztacalco
08500 México, D. F.
558-2277

GILLETTE DE MEXICO Y CIA.
Mariano- Escobedo No. 165
Col. Anáhuac
Deleg. Miguel Hidalgo
México, D. F.
545-6920 ext. 175

39. ANICETO MORALES GIL
Cerro Macuiltepec No. 445
Col. Churubusco
Deleg. Coyoacán
04200 México, D. F.
549-4273

CONTROL VEHICULAR ELECTRONICO, S.A.
Calle 5 No. 187
Col. Pantitlán
Deleg. Iztacálco
08100 México, D. F.
763-7222

... 6

40. J. ANTONIO NAVARRO NIETO
Miramon No. 184-7
Col. M. Carrera
Deleg. Gustavo Madero
México, D. F.

INSITUTO MEXICANO DE COMERCIO EXTERIOR
Alfonso Reyes No. 30
Col. Condesa
Deleg. Miguel Hidalgo
06140 México, D. F.
577-1044 ext. 398

41. ADMON. DE PROYECTOS EN
INFORMATICA

GENERAL MTORS DE MEXICO
Lago Victoria No. 74
Col. Ampliación Granada
México, D. F.
254-3777

42. JOSE LUIS ORTEGA ARRIAGA
Av. Escalera No. 300 E. 4
Ent. A-002
Col. San José La Escalera
Deleg. Gustavo A. Madero
México, D. F.

TANDEM COMPUTERS DE MEXICO
Av. Juárez No. 14 2° Piso
Col. Centro
Deleg. Cuauhtémoc
06050 México, D. F.
585-8688

43. ENRIQUE OSORIO SARMIENTO
Calle Paz Montes de Oca
No. 31 - 210
Col. General Anaya
Deleg. Benito Juárez
03330 México, D. F.
687-1027

BANCO DEL ATLANTICO
Insurgentes Sur No. 774
Col. Del Valle
Deleg. Benito Juárez
México, D. F.
518-7500 ext. 1956

44. CARLOS PARADA PLASCENCIA
Av. Pedregal No. 27-A
Col. Frac. Lomas de Cantera
Deleg. Naucalpan
Edo. de México
576-3164

BANCO SOFIMEX SNC.
Bolivar No. 18 3° Pis
Col. Centro
Deleg. Cuauhtémoc
06000 México, D. F.
585-2133 ext. 152

45. LEANDRA SILVIA PAZ FARELAS
Río Bomba No. 675
Col. Lindavista
Deleg. Gustavo A. Madero
México, D. F.

UNIVERSIDAD DEL VALLE DE MEXICO
San Juan de Dios No. 6
Col. Huipulco
México, D. F.
674-1400

46. ALEJANDRO PEÑA AYALA
Calle 9 Manz. 103 No. 1099-A
Col. Ejido de Iztapalapa
Deleg. Iztapalapa
09310 México, D. F.
686-1793

INSTITUTO MEXICANO DE COMERCIO EXT.
Alfonso Reyes No. 30
Col. Condesa
Deleg. Cuauhtémoc
México, D. F.
211-0036 ext. 273

47. JORGE R. POBLETE MUÑOZ
Uxmal No. 56-303
Col. Narvarte
Deleg. Benito Juárez
03020 México, D. F.
519-6653

INSTITUTO MEXICANO DE COMERCIO EXT.
Alfonso Reyes No. 30
Col. Condesa
Deleg. Cuauhtémoc
06140 México, D. F.
211-0036 ext. 132

48. RICARDO RANGEL MONROY
Jose Ma. Ftrz Rojas 184
Col. Colmena
Deleg. Iztapalapa
09180 México, D. F.
590-3152

SECRETARIA DE DESARROLLO URBANO Y ECOLOGIA
Av. Universidad s/n frente a Mítla
Col. Narvarte
Deleg. Benito Juárez
03020 México, D. F.
590-3152

49. ABRAHAM RIVERA IBARRA
Romita No. 18-2
Col. Roma
Deleg. Cuauhtémoc
06700 México, D. F.
525-7371

INSTITUTO MEXICANO DE COMERCIO EXT.
Av. Alfonso Reyes No. 30-6
Col. Condesa
Deleg. Cuauhtémoc
06140 México, D. F.
211-0036 ext. 150

50. SERGIO ROJAS JUAREZ
Calle Itzapán Manz. 443 Lote 60
Col. Cd. Azteca 3a. Secc.
Ecatepec, Edo. de México
391-1109 755-2800

AUTO TRANSPORTES TRES ESTRELLAS DE ORO S.A. DE C. V.
Calzada Vallejo No. 1268
Col. Sn Juan Ixtacala
México, D. F.
391-1109

51. MARIO ROMERO SALGADO
18 de Marzo No. 10
Col. Sn Miguel
Deleg. Iztapalapa
México, D. F.

IND. RESISTOL, S. A.
Camino al Lago de Guadalupe No. 59
Col. Lechería
59400 México, D. F.
565-4711

52. VICTOR ALEJANDRO SALAZAR SOTO
Codorniz No. 41
Col. Las Alamedas
Deleg. Atizapán
México, D. F.

GENERAL MOTORS DE MEXICO, S.A.
Lago Victoria No. 74
Col. Ampliación Granada
Deleg. Hidalgo
México, D. F.
254-3777 ext. 408

53. JORGE HUMBERTO SALCIDO SALCIDO
Filosofía y Letras No. 21-302
Col. Copilco
Deleg. Coyoacán
04360 México, D. F.
523-0245

URANIO MEXICANO
Insurgentes Sur No. 1079
Col. Noche Buena
Deleg. Benito Juárez
México, D. F.
563-7995

54. ANTONIO SALDAÑA GALICIA
Emiliano Zapata No. 23-F
Col. Sta. Ma. Tomatlán
Deleg. Iztapalapa
México, D. F.
670-9468

TELEFONOS DE MEXICO, S.A.
Melchor Ocampo No. 257-501
Col. Anzures
Deleg. Cuauhtémoc
México, D. F.
592-2523

55. FERNANDO SALINAS AGUIRRE
Chapala No. 16
Col. La Laguna
Sta. Clara, Edo. de México
569-5025

S.C.T. DIRECCION DE AUTOTRANSPORTE FED.
Calz. de las Bombas No. 411
Col. Sta. Ursula
Deleg. Coapa
México, D. F.

56. JOSE ANTONIO SANCHEZ RAMIREZ
Pedro Loza No. 416
Guadalajara, Jal

AUTOTRANSPORTES TRES ESTRELLAS DE ORO
Calz. Vallejo No. 1268
Col. San Juan Iztacala
México, D. F.
391-1109

57. LUIS FELIPE SANSORES RAMIREZ
Ruben Dario No. 47
Col. Moderna
Deleg. Benito Juárez
03510 México, D. F.
590-5537

INSTITUTO MEXICANO DE COMERCIO EXT.
Alfonso Reyes No. 30
Col. Condesa
Deleg. Cuauhtémoc
06140 México, D. F.
211-0036 ext. 132

58. RAUL LAZARO SERRANO LOPEZ
Calle 26 Manz. 15 Lote 5
Col. Rodeo
Deleg. Iztacalco
México, D. F.
558-1529

INSTITUTO DE INVESTIGACIONES ELECTRICAS
Dante No. 36 8° Piso
Col. Anzures
Deleg. Cuauhtémoc
México, D. F.

59. ULISES SERRATO SALA
José Ma. Correa No. 264-1
Col. Viaducto Piedad
Deleg. Ixtacalco
08200 México, D. F.

GRUPO TOLTECA
Torres Adalid No. 527
Col. Del Valle
México, D. F.
687-2030 ext. 168

60. GILBERTO TOVAR

S.C.T

61. MARTE TREJO SANDOVAL
Canada No. 180 3A
México, D. F.

P.V.C.
Carmona y Valle No. 11
Col. Doctores
Deleg. Cuauhtémoc
México, D. F.
761-3811

62. DANIEL VAN DER MEER ORTIZ
Coapa No. 238-104
Col. Toriello Guerra
Deleg. Tlalpan
01450 México, D. F.

INSTITU TO MEXICANO DE COMERCIO EXT.
Alfonso Reyes No. 30
Col. Condesa
Deleg. Cuauhtémoc
06140 México, D. F.
286-0579

63. MA. DEL SOCORRO VARGAS VERA

64. FERMIN IGNACIO VAZQUEZ VAZQUEZ
Luis G. Inclán No. 2805
Col. Villa de Cortés
Deleg. Benito Juárez
03530 México, D. F.
590-5642

APISA GRUPO ICA
Viaducto No. 81
Col. Tacubaya
Deleg. M. Hidalgo
11870 México, D. F.
277-3599

65. ADALBERTO VELAZQUEZ MENDEZ
Ahorro Postal No. 63-44
Col. Postal
Deleg. Benito Juárez
03410 México, D. F.
523-9964

S.P.P.
Insurgnetes Sur No. 795
Col. Nápoles
México, D. F.